# Filtrage et réduction de tests combinatoires

Taha Triki

UNIVERSITÉ DE GRENOBLE

# THÈSE

Pour obtenir le grade de

## DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006


Présentée par

## Taha TRIKI

Thèse dirigée par **Mr Yves Ledru** et
codirigée par **Mme Lydie du Bousquet**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**


# Filtering and reduction techniques of combinatorial tests

Thèse soutenue publiquement le **4 Octobre 2013**
devant le jury composé de :

**Jean-Claude Fernandez**
Professeur, Université Joseph Fourier, Président
**Fatiha Zaïdi**
Maître de conférences, Université Paris-Sud XI, Examinateur
**Fabrice Bouquet**
Professeur, Université de Franche-Comté, Rapporteur
**Virginie Wiels**
Maître de recherche, Laboratoire Onera, Rapporteur
**Yves Ledru**
Professeur, Université Joseph Fourier, Directeur de thèse
**Lydie du Bousquet**
Professeur, Université Joseph Fourier, Co-directrice de thèse





*Université Joseph Fourier / Université Pierre Mendès France /*
*Université Stendhal / Université de Savoie / Grenoble INP*

# Acknowledgments

I would like to express a deep appreciation to my advisors Yves Ledru and Lydie du Bousquet for generously providing me the help and the encouragement continuously during the three years of my thesis. They have always pushed me to my limits to make it possible the successful completion of this manuscript.

I would like to thank all my research colleagues, especially German Vega, Karen Angélica, Kalou Cabrera, Julien Botella and Frédéric Dadeau, that were willing to give me the technical support when I need it to resolve the problems I encountered in the development of my thesis.

I also thank my friends, who have encouraged me by a word or a hand to complete this thesis, especially Ghazi Gharbi, Azzedine Amiar, Amin Ben Sassi, Mohamed Smati.

For you, my wonderful God gifts, Mom and daddy, the deepest gratitude. You helped me financially and spiritually to accomplish my thesis in a kind and successful way. I don't have the suitable words that can express my gratitude. I will just say I love you so much.

For you my darling, a special gratitude, you were always spiritually near to me, giving to me a real strength by hearing your romantic words, especially in my difficult moments.

Above all, all the thank is for our God the most merciful for all his graces, he provides me the intellect and the strength to provide this work.

## Filtering and reduction techniques of combinatorial tests

**Abstract:** The main objective of this thesis is to provide solutions to some combinatorial testing issues. The combinatorial testing consists in generating tests that cover all combinations of defined input values.
The first issue of this thesis is that combinatorial testing can generate a large number of tests that are invalid according to the specification of the System Under Test (SUT). These invalid tests are typically those which fail the precondition of system operation. These invalid tests must be discarded from the set of tests used to evaluate the SUT, because they lead to inconclusive verdicts. As a solution, we propose to couple the combinatorial testing technique to an animation technique that relies on a specification to filter out invalid tests.
In our work, combinatorial tests are generated from a test pattern. It is mainly defined as a sequence of operation calls, using a set of values for their parameters. The unfolding of a complex test pattern, where many operation calls and/or input values are used, may be subject to combinatorial explosion, and it is impossible to provide valid tests from the test pattern. This is a second issue of this thesis. As a solution, we propose an incremental unfolding and animation process that allows to filter out at early stage (in the operation sequence) invalid tests, and therefore to master the combinatorial explosion. Other mechanisms of filtering are proposed to filter out tests which do not cover some operation behaviors or do not fulfill a given property.
The test suites generated from a test pattern can be very large to execute on the SUT due the limited memory or CPU resources. This problem is defined as the test suite reduction problem, and it is the third issue of this thesis. As a solution, we propose a new test suite reduction technique based on annotations (called tags) inserted in the source code or the specification of the SUT. The execution/animation of tests generates a trace of the covered annotations. Based on the trace, a family of equivalence relations is proposed, to reduce a test suite, using the criteria of order and number of repetition of covered tags.
**Keywords:** Combinatorial testing, Test suite reduction

# Contents

Chapter 1

# Introduction

## Contents



## 1.1 Introduction to Software testing

Software is a crucial component in all modern systems such as computers, cars, planes or aircrafts. It is the spirit that gives life to the devices to provide their functions as specified by the system designer. The users assume that a system will always behave as they expect; nevertheless it is not the case. The system can present to the user unexpected outputs with respect to the software requirements. These unexpected outputs arise from faults in the source code. A *fault* is defined as a static defect in the software, committed by systems developers. Executing the software, the fault is manifested to produce in incorrect internal state called an *error*. This error generates an external incorrect behavior, with respect to the requirements or some other description of the expected behavior, called a *failure* [Laprie 1992].

The manifestation of faults in critical systems can cause huge loss and sometimes causes death. As examples of world-known problems caused by software failures, we mention:

- The Ariane 5 rocket explosion in 1995 due to a failure in the inertial reference system. Specifically, it consists in a failed conversion of floating number from 64 bits to 16 bits. This failure caused a loss of more than US$370 million[1].

[1] http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

- The Toyota brakes problem of 2010 due to a fault in the anti-lock braking (ABS) software[2].
- The THERAC-25 radiation machine software failure. Six accidents happened causing three different injuries and three dead [3].
- A fault in the Airbus 319 Safety Critical Software Control. It caused a loss of the autopilot, the navigation displays, and the radio for two minutes[4].
- The US Northeast Blackout of 2003, affected an estimated 10 million people in Ontario and 45 million people in eight U.S. states. It caused financial losses of $6 Billion USD. This is caused because the alarm system in the energy management system failed due to a software error[5].

Given the danger that can cause these faults in modern systems, finding all of them becomes a real challenge. To find faults in a developed software, one technique used by software engineers is *testing*. It is the primary technique used in industry to evaluate a software by observing its execution [Ammann 2008]. It becomes an important activity in software development cycle to evaluate a developed artifact and to assess its reliability. It can reach 50% of the total development budget [Yang 2008].

Before presenting our research work in testing, we begin by presenting the testing activity.

We call the evaluated system a System Under Test (*SUT*). The testing activity as defined by Ammann and Offutt consists in four sub-activities performed by a *test engineer* [Ammann 2008, Ammann 2010]:

1. Designing a test (or a test case): *"A test case is defined as a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly"*[6]. Designing a test case consists in designing test values used to evaluate the SUT. It can be human-based designing, by using domain knowledge of the program and human knowledge of testing. It can be criteria-based by designing values aiming to satisfy coverage criteria or other engineer goals. The coverage criteria can be used as stopping rule to tell whether the set of test cases is adequate for testing, and whether we need to design other test cases.

[2] http://en.wikipedia.org/wiki/2009%E2%80%9311_Toyota_vehicle_recalls#Anti-lock_brake_software_recall
[3] http://sunnyday.mit.edu/papers/therac.pdf
[4] http://www.therazor.org/?p=979
[5] http://en.wikipedia.org/wiki/Northeast_blackout_of_2003
[6] http://softwaretestingfundamentals.com/test-case/

2. Test automation: The tests designed are translated in a computer-language to executable test cases by defining the platform and technologies needed to execute them. In some cases, the test automation is not necessary to evaluate an SUT. After designing a test we can execute it manually by entering to the system the input values needed to perform some functionalities, and observe the behavior of the system.

3. Test execution: It consists in running the executable tests and recording their results.

4. Test evaluation: It consists in evaluating the results of testing and establishing a report for developers. To decide whether a test fails or not, we have to know what we expect after its execution. We can expect for example that the test should not trigger any system exception. A test can also be considered as failing if it does not provide the expected results or outputs. These can be related to some test requirements or to the specification of the SUT. We call the mechanism that tells whether a test fails or not a *test oracle*.

In this thesis, test design and automation are based on combinatorial generation. Test evaluation uses assertion based specification, expressed in OCL or JML to decide on the success/failure of test execution. Test execution corresponds to the execution of Java programs or the animation of UML/OCL specification.

In next section we introduce the main testing technique used in our research work: the combinatorial test generation. We present the motivation of using it, its principle and a small example to illustrate the technique.

## 1.2 Combinatorial testing

The exhaustive testing is the only testing technique that detects all the failures in software with respect to a given test oracle. It consists in executing the system by trying all possible combinations of input values in every possible system state. However, this technique is not used in practice because it is either very expensive or impossible to implement, due to the infinite or the large number of values of the input domain or of the system states.

Therefore, to test a system, relevant values are chosen that are likely to detect errors. These values can be selected by a human (test engineer) that has a knowledge about the specification of the system. The selection of values can be performed using some techniques such as boundary value analysis technique [Jeng 1994]. This technique consists in selecting boundary values

from partitions of values as representative ones. The values can also be chosen randomly.

A test under design may contain many inputs. For instance, let us consider that we aim to design a test that performs a single call to an operation having several parameters. For every parameter, a set of relevant values can be considered to test the operation. Designing one test consists in choosing a single value for every parameter.

Designing a set of test cases that covers all combinations of (some) input values is called *combinatorial testing* [Kuhn 2010]. In our research work we have been interested in such testing technique to evaluate a SUT. The advantage of using this technique is that it allows to generate a large number of tests with minimum manual effort (by using a combinatorial tool). It requires for the test engineer a minimum knowledge about the specification, for example knowledge about the interfaces of operations. Moreover it is intended to explore systematically system behaviors by combining different input values. One can see how the system will behave with interaction of some input values, for which the test may be invalid according to the specification.

Exhaustive combinatorial testing allows to generate all combinations of input values. Other researches consider that exhaustive combinatorial testing can be expensive and it has been shown that most failures are caused by interactions between few parameters [Kuhn 2010]. Therefore, it has been proposed that generating tests that cover all pairs of parameters values is sufficient. The generalization of the technique is the generation of n-combinations of parameter values. It aims at finding failures caused by interaction of n input parameters. The mentioned combinatorial testing techniques are implemented in combinatorial tools. For example the AETG is a tool that generates a test set that covers the n-combinations of input values defined by the user.

Let us illustrate the combinatorial testing by an example. Consider that our SUT is a container manager. It has 4 containers `lo1`, `lo2`, `lo3`, `lo4` initialized as empty. The system has a method `load (int c1, int c2, int c3, int c4)` that loads the containers by number of kilograms (defined as integer). It adds the load value (positive value) given in c$i$ to cont$i$ (with $i$=1 to 4). We suppose that every container has a limit of load, `lo1` can be loaded by maximum 1000 kg, `lo2` by 3000 kg, `lo3` by 5000 kg and `lo4` by 7000 kg.

Let now consider that we aim to create tests to evaluate the `load` method. The exhaustive testing is not possible due the large number of values for the input domain (4-tuple of integers = $(2 * Maxint)^4$). Therefore, we select 2 representative values for each parameter. One value less than the maximum load and another one greater than the maximum load, such as using 1000 and 1001 for `c1`. We get `c1`=[1000,1001], `c2`=[3000,3001] ,`c3`=[5000,5001] and

c4=[7000,7001]. Performing one call to the `load` method using the values selected can be represented by the following test pattern:

```
load ([1000,1001], [3000,3001], [5000,5001], [7000,7001])
```

This test pattern defines abstractly the set of test cases that can be generated to test the `load` method.

Using exhaustive combinatorial technique, this pattern generates 16 (2*2*2*2) test cases:

```
TC1: load (1000, 3000, 5000, 7000)
TC2: load (1000, 3000, 5000, 7001)
TC3: load (1000, 3000, 5001, 7000)
...
TC16: load (1001, 3001, 5001, 7001)
```

The pairwise testing[7] (2-way combinations) generates a reduced test set compared to exhaustive combinatorial testing. It generates a subset of tests that covers all pairs of parameter values. The result can be:

```
TC1: load (1000, 3000, 5000, 7000)
TC2: load (1000, 3000, 5000, 7001)
TC3: load (1000, 3001, 5001, 7000)
TC4: load (1001, 3000, 5000, 7000)
TC5: load (1001, 3000, 5001, 7001)
TC6: load (1001, 3001, 5000, 7001)
```

In our research work, we have used the Tobias tool [Ledru 2004] that is an exhaustive combinatorial testing tool developed by our research team. It takes as input a test pattern and unfolds it combinatorially into a possibly large set of test cases. The test pattern describes abstractly a test case using many constructs. For example we can define a test pattern that performs a sequence of operation calls using a set of values for their parameters. Such as using a set of values for an operation parameter, we can also define a set of operation calls at some point in the test pattern. Other constructs can be used in the Tobias test pattern as iterating an operation call. To illustrate the definition of test pattern in Tobias let us consider that the SUT (container manager) has another operation `unload (int num, int lo)` to unload the containers. The `num` parameter specifies the number of the container

[7] We used the website http://alarcosj.esi.uclm.es/CombTestWeb/combinatorial.jsp to generate the exhaustive combinations and the pairwise combinations (using AETG algorithm).

to unload (must be from 1 to 4). The `lo` parameter takes the number of kilograms to unload (positive value, less then or equal to the load of the corresponding container). Let us consider the following `Load1To2ThenUnload` Tobias test pattern:

```
Load1To2ThenUnload:
ContainerManager contManag = new ContainerManager();
contManag.load ([500,300], [3000,1], [3000,3500], [7000,0]){1,2};
contManag.unload([1,2,3],4000) | unload ([4,5], 8000);
```

It creates an instance of the SUT (`ContainerManager`), loads into containers valid numbers of kilograms and iterating the `load` operation call from one to two times (`{1,2}`). It means that the test will begin by either one call or two calls to `load` operation. Next, the test pattern performs either a call to `unload` using the set of values 1, 2 and 3 for the `num` parameter and 4000 for the `lo` parameter, or a call to unload using the set of values 4 and 5 for the `num` parameter and 8000 for the `lo` parameter.

The test pattern `Load1To2ThenUnload` generates 1360 tests $= (16^1 + 16^2) * (3 + 2)$. We present the first and the last ones in the following:

```
TC1: ContainerManager contManag = new ContainerManager();
     contManag.load(500,3000,3000,7000) ;
     contManag.unload(1,4000) ;
...
TC1360: ContainerManager contManag = new ContainerManager() ;
        contManag.load(300,1,3500,0) ;
        contManag.load(300,1,3500,0) ;
        contManag.unload(5,8000) ;
```

These combinations of values allow to test the system using diverse interesting scenarios. In next sections we present the issues of this thesis related to combinatorial testing for which solutions have been proposed.

## 1.3 Model-based filtering

The combinatorial testing technique can generate a very large number of tests from few lines of pattern definition. However, they do not rely on a specification to perform this generation. Therefore, a large number of generated tests will be illegal according to the specification and their execution will result in

inconclusive verdict. We call this kind of test an *invalid test*. For example a test that contains an operation call that fails the operation precondition is an invalid test. These invalid tests lead to an inconclusive verdicts and do not tell any useful information about the SUT. They have to be discarded from the generated test set used to evaluate the system.

For example, in the 1360 tests generated from `Load1To2ThenUnload` test pattern there are only 48 tests that are valid according to the specification of the container manager[8]. The other 1312 tests are invalid because the test tries to

- load a number of kilograms that exceeds the limit of a container or/and
- unload a number of kilograms from a container greater than the available one or/and
- unload a number of kilograms from an inexistent container

The 1312 invalid tests have to be discarded from the generated test set because they do not satisfy the specification and can not be candidate to test the SUT.

The first issue of this thesis for which we propose a solution is the problem of invalid tests generated from a combinatorial unfolding. The solution we propose consists in using an animatable or executable specification when executing the test cases. The execution of the specification detects invalid test cases.

However, for complex test patterns where, for example, many input values are defined, the test generation is subject to combinatorial explosion. It consists in generating from a test pattern a huge number of tests. The complete test pattern unfolding or the animation of all generated tests is impossible to perform in such case due the limited computer resources. For example, the following pattern uses larger sets of input values:

```
LoadmThenUnloadn:
ContainerManager contManag = new ContainerManager();
contManag.load([58,302,450,605],
               [3210,1000,40,0],
               [3000,2500,4123],
               [3501,0]){2};
contManag.unload([1,2,3,4,5],[1523,8542,321,789,672,259]){3};
```

Unfolding the test pattern `LoadmThenUnloadn` results in $2.48832 * 10^8$ tests which exceeds the capability of the Tobias tool (about $10^6$ test cases).

[8]The specification is defined using JML language embedded in the Java implementation of the SUT intended to be checked at runtime (see Appendix A).

Therefore, the second problem addressed by our model based filtering proposes a solution to deal with the problem of combinatorial explosion. In a test pattern, two cases contribute in the combinatorial explosion of the test pattern:

- The combination of values used in one instruction. For example for the schema `LoadmThenUnloadn`, let us consider that the first instruction is to load the container one time using some values. The combination of values consists in combining the values used in the parameters of `load` method (4*4*3*2=96).

- The combination of values between instructions. The repetition of the `load` operation 2 times is a combination of values between two instructions (two calls to the `load` operation). The $2.48832 * 10^8$ tests generated from `LoadmThenUnloadn` pattern are computed by multiplying the number of elements unfolded from the two calls to `load` operation ($96^2$=9216) by the number of elements unfolded from the three calls to `unload` operation ($30^3$=27 000).

To reduce the number of combinations in the first case, the test engineer can reduce the number of combinations by reducing the number of values used in the parameters of the `load` operation call.

We consider the second case an important factor making the test pattern explosive and is a main issue of our thesis. The solution proposed for this issue is to incrementally unfold the test pattern. It means that test pattern will be processed instruction by instruction. We consider the first instruction of the test pattern, we unfold it and we animate the generated tests to get the valid ones. We use then the valid ones as prefixes to be combined with the next instruction. This incremental process is done until all instructions are processed. The advantage of performing this is the deletion of the invalid prefixes at early stage, and the number of combinations between two instructions will decrease. It becomes possible to process a test pattern with billions of tests and to get final valid tests.

The test pattern can be rewritten as:

```
LoadmThenUnloadn:
1 ContainerManager contManag = new ContainerManager();
  contManag.load ([58,302,450,605],
                  [3210,1000,40,0],
                  [3000,2500,4123],
                  [3501,0]);
2 contManag.load ([58,302,450,605],
```

```
                    [3210,1000,40,0],
                    [3000,2500,4123],
                    [3501,0]);
3 contManag.unload([1,2,3,4,5],[1523,8542,321,789,672,259]);
4 contManag.unload([1,2,3,4,5],[1523,8542,321,789,672,259]);
5 contManag.unload([1,2,3,4,5],[1523,8542,321,789,672,259]);
```

The first unfolding iteration consists in unfolding the `load` method in 96 calls. Only 72 calls are reported as valid. The second iteration consists in combining the 72 valid prefixes with 96 elements of the second instruction. This unfolds in 6912 elements. We report that only 351 tests are valid and they are combined with the third iteration to be unfolded in 10 530 (351*30) test cases. The incremental process continues until treating all instructions in the schema mechanisms. When the number of remaining valid test cases is too large, we propose additional filtering to keep a proportion of the valid tests, or those which satisfy a given property. Using our filtering approach for `LoadmThenUnloadn` test pattern, we finally get 6690 valid test cases out of $2.48832 * 10^8$.

We call this first contribution *model-based filtering of combinatorial tests*. It was published in the FASE (Fundamental Approaches to Software Engineering) international conference of 2012.

In next section we present the second contribution of this thesis.

## 1.4 Trace based reduction

The tests unfolded from test patterns and reported as valid according to the specification are used to evaluate the implementation of the SUT. However, The number of valid tests can also be too large or difficult to execute due the limited memory or CPU resources of the SUT. Moreover, in our research work, tests generated can be used to evaluate applications embedded in smart cards that have very limited resources. Additionally, in the context of regression testing, many tests are added to the test suite (a suite/sequence of test cases) to evaluate new or modified requirements and thus the test suite becomes large and the cost of executing it becomes expensive. A second issue of this thesis is to study the reduction of these large test suites.

The objective is to run a subset of tests rather than the original test suite. The challenge here is to generate a reduced test suite that is representative of the original one in terms of fault detection capability.

The test suite reduction problem was originally studied by Harrold et al [Harrold 1993], and was later addressed by numerous authors [Lin 2009, Sprenkle 2005, Parsa 2009]. Two big families of approaches are reported in

literature performing the test reduction: the coverage based approaches and the similarity based approaches. The coverage based techniques use the structural coverage information result from test execution to reduce the test suite. For example, if a test suite contains 3 tests **ta**, **tb** and **tc**. Assume that test **ta** covers a method branch **mb1**, test **tb** covers a method branch **mb2** and test **tc** covers both branches **mb1** and **mb2**. In the reduced test suite we keep only **tc** because it covers all branches covered by **ta** and **tb**. This was the idea of Harrold, she developed an heuristic called HGS to reduce a test suite based on their coverage information (e.g. branch coverage, statement coverage). The similarity based techniques use a similarity function or an equivalence relation to state equivalence between tests and reduce the test suite. When several tests of the test suite are equivalent, only one of them is kept in the reduced suite. For example, in [Masri 2007], the authors propose a similarity-based approach to select test cases based on their execution trace.

In our research work, we proposed a new test reduction technique that reduces a test suite using an equivalence relation, based on traces generated from test execution. Our approach relies on annotations (called tags) inserted in the source code or in the specification of the SUT. They are intended to trace user requirements or to instrument the code. These tags are covered during the execution/animation of tests. Using the criteria of order and repetition of tags in the execution of the test case, a family of equivalence relations are defined. The weakest relation does not take into account the order and the number of repetition of tags covered to compare two test cases. The strongest relation requires that the traces of equivalent test cases have the same sequence of tags.

Let us illustrate our test reduction solution on the SUT container manager. We suppose that the SUT has another `loadC4` method that loads container `cont4` (`cont`) using an array of values (`int [] vals`). It accepts positive and negative integers and loads the absolute value. We suppose also that the container `cont4` has no more a limit of load. The Java code of the `loadC4` method is presented as follows:

```
public class ContainerManager {
  private int cont1=0;
  private int cont2=0;
  private int cont3=0;
  private int cont4=0;

  ...// code of load and unload methods

  public void loadC4(int [] vals){
```

```
    for (int i = 0; i < vals.length; i++) {
      int x = vals[i];
      if (x > 0) {
        cont4 = cont4 + x;
        log("load-gt0");
      }
      else if (x < 0) {
        cont4 = cont4 - x;
        log("load-lt0");
      }
      else {
        log("noload");
      }
   }
  }
}
```

The tags are inserted in the `loadC4` method branches in the form of `log("tag")`. The method `log` allows to trace a tag and its execution order in the operation and in the test. Now, let us consider the following test suite TS used to evaluate the IUT `cm`:

```
T1: cm.loadC4(new int{}[0,5000,-1500]);
T2: cm.loadC4(new int{}[1500,-200]); cm.loadC4(new int{}[0]);
T3: cm.loadC4(new int{}[1400,-900]); cm.loadC4(new int{}[0]);
```

The weakest equivalence relation considers that all tests in TS are equivalent because all of them cover the set of tags: `load-gt0`, `load-lt0` and `noload`, even if they have for example different size (number of operation calls) such as T1 and T2. If we reduce TS according to this relation we will have a reduced test suite with one test (selected randomly), for example T1. The strongest equivalence relation considers that two equivalent tests have to trace the same sequence of tags. For example T2 and T3 cover exactly the same sequence of tags: `load-gt0`, next `load-lt0` and next `noload`. TS is reduced using this equivalence relation into two tests, for example T1 and T3. We give only 2 equivalence relations to illustrate the principle of our test reduction, but we proposed 2 others that will be detailed in this thesis.

We called this second contribution *test suite reduction using equivalence relations based on code annotations*. It was published in the AFADL (Approches Formelles dans l'Assistance au Développement de Logiciels) French conference of 2011 .

Our research work was part of the ANR TASCCC project oriented to test applications embedded in smart cards. The two contributions were experimented on two case studies provided by our project partners: the Global Platform case study a last generation operating system for smart cards, and on-line vending system of cinema tickets. In these case studies, these test patterns are generated automatically from test properties [Castillos 2011]. We also used our contribution in other applications such as the electronic purse application, a case study used in our team. Using our contributions in these case studies and examples show how they can efficiently resolve reduction and filtering problems.

This thesis is organized as follows:

- The first part of this thesis presents the: model-based filtering of combinatorial tests.
- The second part presents the: test suite reduction using equivalence relations based on code annotations.
- Finally we present a conclusion to this thesis.

# Model-based filtering of combinatorial tests

Chapter 2

# Introduction

Contents



## 2.1 Motivation

In our research work we have been interested by combinatorial test generation techniques and especially by the combinatorial testing tool Tobias developed by our research team. The use of a combinatorial testing technique for test generation is motivated by the fact that it generates a large number of test inputs with minimum effort. The generated tests are similar according to some *pattern*, for example the operation name or the operation sequence, but diverse in terms of the parameters values. These defined values are generally chosen by a human or an automatic system, that considers them relevant to observe a large number of system behaviors.

However, combinatorial test suites may lead to a large proportion of invalid test cases, i.e. test cases which will not conform to the specification. These invalid test cases should be removed from the test suite because they correspond to illegal inputs or sequence of calls. Their executions result in inconclusive verdicts. Removing invalid test cases is especially interesting if the test pattern is complex or if many input values are used. The number of combinations can increase rapidly up to billions of test cases. It is impossible to consider so large test suites because they require intractable resources. A very large test suite might be impossible to compile and execute using the standard compilers, test drivers and computer resources.

An idea is to rely on a specification to filter out invalid test cases at early stages in the unfolding process. Doing that will help fighting the combinatorial explosion.

## 2.2 Contribution

In order to discard invalid test cases during the combinatorial unfolding, we have implemented a model-based testing approach where Tobias test cases are first run on an executable specification. This animation of test cases on a model allows filtering out invalid test sequences produced by blind enumeration, typically the ones which violate the pre-conditions of operations. To do that, we introduce extensions in Tobias tool which support an incremental unfolding and filtering process. Moreover, we added several constructs which help the test engineer expressing more precise test patterns and allow to filter out valid test cases which do not meet the intent of the test pattern. These new constructs could mandate test cases to satisfy a given predicate at some point or to feature a given behavior.

The early detection of invalid or unintended test cases improves the calculation time of the whole generation and execution process, and helps fighting combinatorial explosion.

This part of the thesis is composed of 4 chapters:

- Chapter 3 presents the combinatorial testing principle, the different combinatorial techniques and tools reported in literature especially the combinatorial testing tool Tobias a main tool in our research work.
- Chapter 4 details our approach to filter combinatorial test cases based on specification, and related research works.
- Chapter 5 presents illustrations and limitations of our approach using some case studies.
- Chapter 6 gives solutions to deal with the problems found in our approach, and gives illustrations of these solutions using the case studies.
- Chapter 7 presents a summary of our contribution.

Chapter 3

# Combinatorial Testing

## Contents



## 3.1 Motivation

Exhaustive testing is a technique that executes the system with all possible combinations of input values [Marinov 2003]. Using this technique, it is intended to observe all possible behaviors of a system, and thus to detect all the implementation failures, with respect to a given test oracle. This technique can be applied when the state space of the SUT is relatively small. For example, consider a stateless web page with a choice list and two buttons. Exhaustive testing of this web page consists in observing the behavior of the system after clicking on each button for each element in the list, i.e. if we have 3 elements in the list we have to try 6 combinations.

In practice, exhaustive testing is infeasible because systems are often much more complex than the example previously presented. For instance, there can be an infinite or huge number of input parameters values or internal system states.

Therefore, one solution consists to select relevant values for testing based on the specification or on the source code of the SUT. The selection can be performed manually by a human that has a knowledge about the system

specification. Some techniques can be used to select relevant values such as boundary testing. The boundary testing or the boundary value analysis is a selection technique based on three steps [Jeng 1994]. First, a technique of equivalence partitioning is applied to classify the inputs into different equivalence classes. Second, the neighborhood, a mathematical technique is applied to detect the boundaries between the equivalence classes. Third, the boundary values on the boundaries of the equivalence classes are selected. This technique considers these boundary values as representative test inputs of the original set of inputs.

After selecting values for each test input, one can create test cases by choosing for each test case a single value for each test input. Covering all combinations of input values is called combinatorial testing. It generates these combinations in different test cases. The advantages of this technique is that it allows to explore systematically system behavior by combining different input values. Moreover, a large number of tests can be generated with a simple-written line.

Different types of combinatorial techniques can be considered and detailed hereafter [Grindal 2005].

## 3.2 Pairwise and n-way testing

The basic form of combinatorial testing is to identify sets of relevant values for system operation parameters, and the generation technique computes *all combinations* resulting in different operation calls. In this case a test case is considered as a single operation call.

Let's take an example of a booking system of a cinema ticket. The booking is allowed depending on the age of the person and the film type. It is performed by the method **book**, which has five parameters. The *filmType* parameter specifies whether the film is restricted by the person age or not. The *3D* parameter tells if the film is projected using the three dimensions technology or not. The *filmName* parameter gives the name of the film to watch. The *age* and *personName* parameters define respectively the age and the name of the film viewer. The signature of the method **book** is defined in Fig. 3.1.

```
public void book (String filmType, boolean 3d, String filmName,
int viewerAge,  String viewerName){...}
```

Figure 3.1: The signature of the method **book**

We associate for each parameter a set of values intended for testing:

- filmType: "-12" ,"-16" or "public"
- 3D: true or false
- filmName: "Die Hard", "Jappeloup"
- viewerAge: 11, 15 or 22
- viewerName: "François", "Nicolas"

Generating an exhaustive test suite from these defined values results in 72 test cases.

In practice, generating all combinations of parameter values to test a single method could be time and resource-consuming. Moreover, it has been suggested that it may be useless to consider all the combinations, since most failures are caused by interaction of relatively few parameters [Kuhn 2010].

For this reason, to reduce the number of generated tests, it has been suggested to consider a subset of tests that covers all pairs of parameter values. It means that every pair of values for two parameters is covered at least one time by a test case. This technique is called the pairwise testing or the 2-way combinatorial testing. Fig. 3.2 presents an example of test set achieving pairwise coverage. They were generated using the AETG on line tool[1]. The number of generated test cases by this tool for this example is 9. The number of tests generated is greater than or equal to $m * n$ where $m$ and $n$ are the numbers of values for each of the two parameters with the largest number of choices (in our example $m$=$n$=3).

| filmType | 3D | filmName | viewerAge | viewerName |
|---|---|---|---|---|
| -12 | true | Die Hard | 11 | François |
| -12 | false | Die Hard | 22 | Nicolas |
| -12 | false | Jappeloup | 15 | Nicolas |
| -16 | true | Die Hard | 15 | Nicolas |
| -16 | true | Jappeloup | 22 | François |
| -16 | false | Die Hard | 11 | Nicolas |
| public | true | Die Hard | 15 | Nicolas |
| public | true | Die Hard | 22 | François |
| public | false | Jappeloup | 11 | François |

Figure 3.2: Pairwise test cases for the cinema ticket booking system

[1] http://alarcosj.esi.uclm.es/CombTestWeb/combinatorial.jsp

The n-way testing is the generalization of the pairwise testing concept(n is the interaction strength). A n-way generation algorithm produces a test set that covers all the n-combinations of parameter values. It aims at capturing failures caused by interaction of $n$ (or less) input parameters [Kuhn 2010]. Some experimentations show that n-way testing decreases the number of required tests (with respect to the exhaustive combinations), while conserving the same fault detection capability [Kuhn 2010].

It is shown by Williams and Probert, that finding the test cases set that achieve n-way coverage can be NP-complete problem [Williams 2001]. Multiple algorithms have been proposed by researchers to generate near-minimum test sets. Cohen et al. [Cohen 2007] classify these algorithms in 3 classes: *Algebraic*, *Greedy* and *Heuristic search*. *Algebraic* solutions use mathematical techniques to ensure a fast production of small covering set. *Greedy* algorithms select new tests in order to cover as many as possible uncovered requirements. *Heuristic search* algorithms apply transformation techniques on a pre-selected set of tests until all combinations are covered.

We can find on-line many other tools[2] performing the pairwise or the n-way combinations generation. For instance, AETG [Cohen 1997], ACTS [Borazjany 2012] and PICT are tools that generate a test set that covers the n-way combinations of the input values defined by the user. These tools use the Greedy algorithm for test generation. Test Cover[3] is example of tool that uses mathematical techniques to generate all-pairs covering array of user defined test parameters. For the example of booking of cinema ticket, PICT generates 21 tests for 3-way interactions and 43 tests for the 4-way interactions. We can see that the number of generated tests increases with the increase of interaction strength.

## 3.3 Extended forms of combinatorial testing

The combinatorial approaches, especially the n-way testing approaches, are intended to generate combinatorial tests considering a *single* method call. Testing a method in isolation of the remaining system methods is not always possible. For instance, let us consider a class under test (in Object Oriented context). A test case includes at least a call to a constructor before being able to call one of its methods.

JMLUnit is an example of tool that takes into consideration this problem [Cheon 2002]. It generates tests that instantiate a Java class by calling the constructor and then call a single method. It allows the user to specify the

[2] http://www.pairwise.org/tools.asp
[3] http://www.testcover.com/

values for the method parameters and generates all combinations in different JUnit tests.

Extended forms of combinatorial testing allow to sequence sets of operations, each operation being associated with a set of relevant parameters values. This produces more elaborate test cases, which are appropriate to test systems with internal memory, whose behaviors depend on previous interactions.

Tobias is one of these combinatorial test generators [Maury 2002, Ledru 2004, Ledru 2007]. It is developed by the VASCO team of the Grenoble Informatics Laboratory. It generates combinatorial tests based on scenarios expressed using regular expressions and a set of operators. It has inspired other combinatorial testing tools, such as the combinatorial facility of the Overture toolset for VDM++ [Larsen 2009] or jSynoPSys [Dadeau 2009].

The next section details the Tobias combinatorial tool.

## 3.4 Tobias tool

### 3.4.1 Principle

Tobias is a combinatorial testing tool developed since 2002 [Maury 2002, Ledru 2004, Ledru 2007]. To generate a test suite, Tobias unfolds a *test pattern* (also called "test schema"). A test pattern describes abstractly a set of test cases, by using a set or sequence of instructions and values.

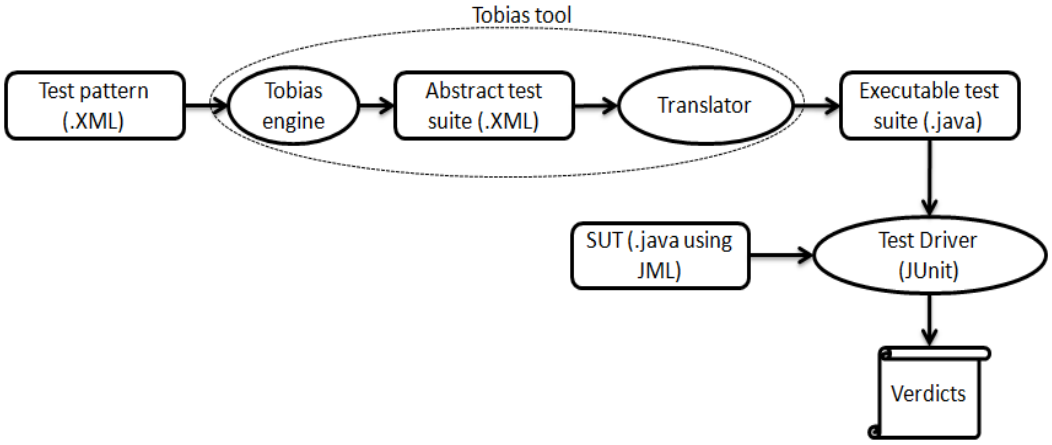



Figure 3.3: Tobias tool principle

Several types of constructs allow the definition of a test pattern in the Tobias input language. The key construct is the **group** construct that is

subject to combinatorial unfolding. It is possible to define a group of operation calls, a group of values, a group of objects, a group of groups etc. Some other constructs can be applied to instructions like iteration or choice. A test schema is unfolded by Tobias in a suite of test cases by computing all possible combinations of elements defined in the groups. Fig. 3.3 gives an overview of the principles of Tobias tool. The process begins by writing a test pattern in the Tobias input language. The test schema is then unfolded into an abstract test suite in a Tobias output language. A file translator tool allows to transform the abstract test suite to an executable one by choosing the target technology (e.g. JUnit test suite). The executable test suite is run on the test driver (e.g. JUnit) using the SUT source code. An additional oracle technology (e.g. JML) might be used with the source code to predict the expected behavior of the system. The test driver provides the verdicts after the execution of tests.

Fig. 3.4 presents an example of a test pattern. The SUT is an electronic purse application that allows the user to manage his bank account (*iut* is the instance under test). An extended version of the electronic purse application will be presented later in Sect. 4.3.1. The abstract test pattern

```
group CreditOrDebitCard [us=true] {
  @checkPinGroup;
  @TransactionGroup{1,2};
}

group checkPinGroup {
  iut.checkPin ([1234,5678]);
}

group TransactionGroup {
  (iut.credit ([0,100]) | iut.debit([50,150]));
}
```

Figure 3.4: Tobias test pattern example

`CreditOrDebitCard` describes a set of test cases that debit or credit a purse (one or two times) after the user authentication. The `us=true` expression indicates that the corresponding group will be unfolded by Tobias into test suite. The user authentication is carried out with `checkPin` operation (checkPin (int pin)) called with correct and incorrect pin values (resp. `1234` and `5678`). `debit` (debit (int val)) and `credit` (credit (int val)) operations are performed each with a group of values (resp. `{50, 150}` and `{0, 100}`). These opera-

```
1: iut.checkPin(1234); iut.credit(0);
2: iut.checkPin(1234); iut.credit(100);
3: iut.checkPin(1234); iut.debit(50);
...
12: iut.checkPin(1234); iut.credit(100); iut.debit(150)
...
39: iut.checkPin(5678); iut.debit(150); iut.debit(50);
40: iut.checkPin(5678); iut.debit(150); iut.debit(150);
```

Figure 3.5: Abstract tests generated from Tobias test pattern

```
public class TS_testSchema{
 @Test
 public void testSequence_1(){
  iut.checkPin(1234) ;
  iut.credit(0) ;
 }

 @Test
 public void testSequence_40(){
  iut.checkPin(5678) ;
  iut.debit(150) ;
  iut.debit(150) ;
 }
}
```

Figure 3.6: JUnit tests generated from Tobias test pattern

tions are defined as choices denoted by `"|"` in `TransactionGroup`. Unfolding the test schema using Tobias results in an abstract test suite given in Fig. 3.5. The actual syntax of these test cases is XML. For readability reasons, we give them in a textual form. It contains 40 test cases: 2 * ((2*2)+ $(2*2)^2$). 8 test cases correspond to an authentication followed by one call to debit or credit operation. 32 test cases correspond to an authentication followed by two calls to debit or credit operations. This test suite is translated to JUnit test suites (given in Fig. 3.6 in JUnit 4 format).

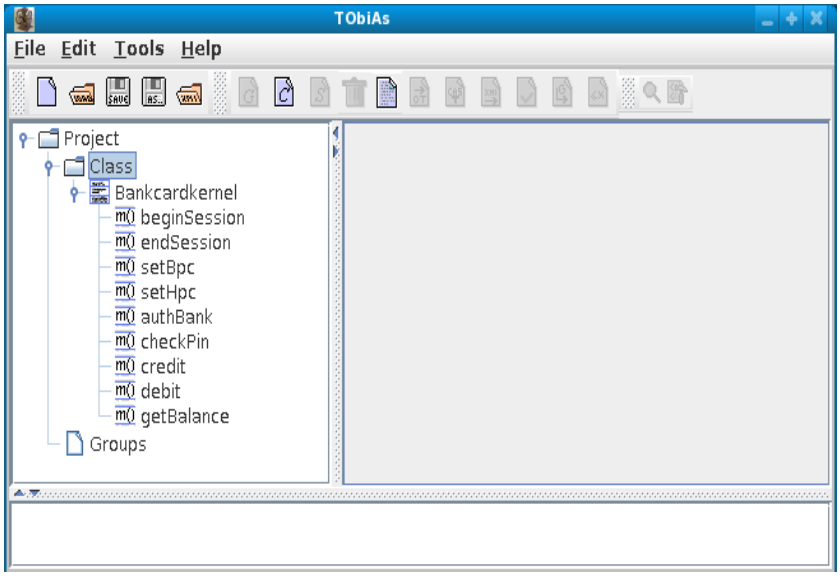



Figure 3.7: Graphical user interface of the first version of Tobias

### 3.4.2 Advantages and Drawbacks of Tobias

Tobias tool ensures a quick generation of large sets of tests and therefore improves the productivity of the test engineer. A 10 line textual description of a schema can be unfolded into thousands of executable tests in less than 5 minutes. Tobias tool is implemented to support the generation of more than one million of test cases from an input test pattern. Moreover, Tobias generates abstract tests, possibly to translate them with multiple target technologies (as Java/JML, C++, VDM, B, UML/OCL). Additionally, many constructs are offered for the test engineer to create complex test scenarios.

Nevertheless, combinatorial testing naturally leads to combinatorial explosion. This is initially perceived as a strength of such tools: large numbers of tests are produced from a test pattern. The size of generated test suites may be a problem when their translation into a target technology such as JUnit, the compilation of the resulting files and their execution need too much computing resources. In practice, the size of the test suite must be limited between 10 000 and 100 000 test cases.

### 3.4.3 A textual language for Tobias

A first version of Tobias was available in 2002 (graphical user interface presented in Fig. 3.7). The constructs available to define a pattern were limited but the test tool was easy to use thanks to a simple textual language and an

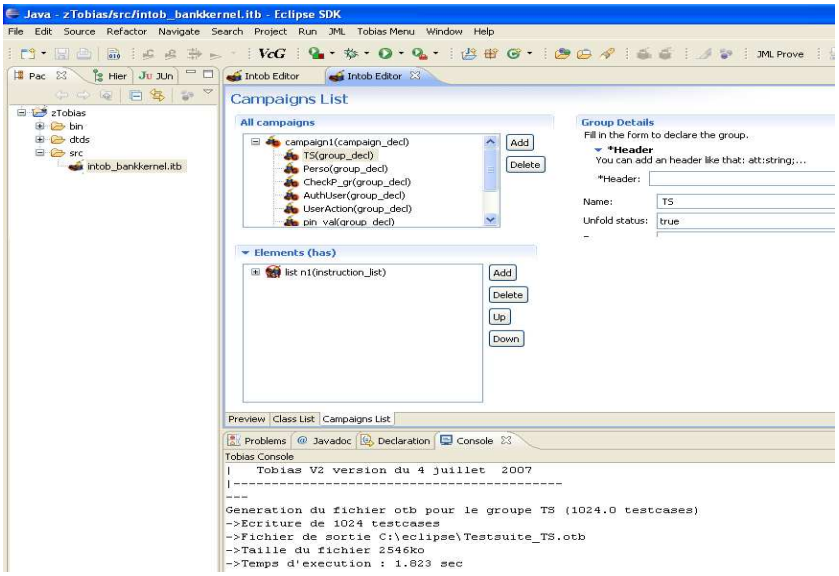



Figure 3.8: Graphical user interface of the second version of Tobias

intuitive interface.

A second version of Tobias was available in 2007 (graphical user interface presented in Fig. 3.8). More constructs were available, but the test pattern should be defined in XML or with the help of a graphical interface. In both cases, it was not a user-friendly task.

This problem was resolved by creating a text input language for Tobias called TSLT (Test Schema Language for Tobias). TSLT is based on the scenario language of the jSynoPSys tool [Dadeau 2009], developed by the LIFC laboratory in Besançon. It has a syntax similar to the syntax of object languages. To be unfolded, a schema defined in TSLT is first translated into a schema in the input language of Tobias thanks to the TSLT compiler. The TSLT is at present the most convenient language to write Tobias test patterns.

### The group construct

A test pattern in TSLT contains a set of group definitions. As we said previously, the group construct is the main construct in Tobias. The group definition in TSLT has the form presented in Fig. 3.9.
The group name (*GroupName*) must be unique. It can be used to refer to the group in other group definitions. It can also be used to name the resulting test suite file (e.g. for JUnit). A group has several attributes among which *us* (unfolding status). It indicates that the corresponding group will be unfolded by Tobias into a test suite, in this case *us* is equal to *true*. In the case

```
group GroupName [ attribute = value , ...] {
  Sequence of instructions
  OR Set of instructions
  OR Set of values
}
```

Figure 3.9: Form of TSLT test pattern

where *us* is equal to *false* (default value), the group is intended to be used in other group definitions. The group definition (body) can be a *sequence of instructions*, a *set of instructions*, or a *set of values*. The body of the group uses a syntax similar to Java. In the following, we present the different types of group body.

### Group of instructions sequence

In the body of this group we define a sequence of instructions separated by a semicolon.

Fig. 3.10 presents a group of instructions sequence. The pattern begins by creating an IUT from the EPurse class. It next credits the purse with the amount of 100 and debits it with the amount of 50. Afterwards, it assigns to a variable x the balance value. It checks in final whether the value of the balance is positive.

```
group PurseSchema [us=true] {
  EPurse ep = new EPurse();
  ep.credit(100);
  ep.debit(50);
  int x = ep.getBalance();
  assertTrue(x >= 0);
}
```

Figure 3.10: Group of instructions sequence in TSLT

Tobias unfolds the test pattern `PurseSchema` into one test case. The combination is performed only when the pattern contains a set of values or a set of instructions.

### Group of values

The group of values is used to define a set of values for an operation parameter that is subject to combinatorial unfolding. Fig. 3.11 presents two possible

```
group PurseSchema1 [us=true] {
  EPurse ep = new EPurse();
  ep.credit(@CreditValues);
  ep.debit([20, 50]);
  assertTrue(x >= 0);
}
group CreditValues {
  values = [50, 60];
}
```

Figure 3.11: Group of values in TSLT

ways to define a group of values in TSLT. The first way is to define a named group of values. For instance, the credit parameter refers to a group of values (named *CreditValues*) containing values 50 and 60. The second way is to define an unnamed group of values. In this case, the set of values is defined directly in the parameter (as for the debit parameter we defined 20 and 50 values). The *PurseSchema1* schema is unfolded to 4 test cases representing the combinations of the two group of values.

### Group of instructions set

The group of instructions set is used to define at a specific point in the schema a disjunction (or choice) between several instructions. In Fig. 3.12, a group of

```
group PurseSchema2 [us=true] {
  EPurse ep = new EPurse();
  @Transaction;
  assertTrue(x >= 0);
}
group Transaction {
  ep.credit(50) | ep.debit(20)
}
```

Figure 3.12: Group of instructions set in TSLT

instructions (named Transaction) defines a choice between a call to credit with

the amount of 50 or a call to debit with the amount of 20. The *PurseSchema2* schema is unfolded into 2 test cases, one test case calls the credit operation and the other one calls the debit operation. Similar to the group of values, the group of instructions can be used directly in the schema without name.

### Iteration

An iteration construct can be used to repeat a set of instructions a specified number of times. In Fig. 3.13, the *credit* operation call is repeated from 1 to 4 times. The iteration construct is denoted in this case by enclosing the lower bound and the upper bound into braces separated by a comma. The *debit* operation call is repeated exactly 5 times. In this case, the iteration construct is denoted by enclosing a fix number into braces.

```
group PurseSchema3 [us=true] {
  EPurse ep = new EPurse();
  ep.credit(100){1,4};
  ep.debit(30){5};
  assertTrue(x >= 0);
}
group Transaction {
  credit(50) | debit(20)
}
```

Figure 3.13: Iteration construct in TSLT

### Summary for TSLT language

Writing a test pattern in the TSLT syntax is easier compared to the XML syntax. It does not require a lot of time to be unlearned. It has been used successfully by master students, and many of our industrial and academic project partners and they found it an easier language for test pattern definition.

The TSLT language was developed during our master thesis [Triki 2010]. In the following, all test pattern examples are written in the TSLT syntax.

In the next section, we present the principle of other techniques used in Tobias called selectors and filters, the motivation of their application, and examples of their definition in TSLT test pattern.

### 3.4.4 Selectors and filters

Some defined test patterns can be unfolded into a huge number of tests. This large unfolding occurs for example when the test pattern contains large sets of input values. The large number of generated tests are impossible to save or to execute due the limited memory or CPU resources. To reduce the size of the generated test suite, Tobias offers to the test engineer the possibility to define filtering and selection mechanisms.

A filter is a property defined by the test engineer that must be fulfilled by a test case. It is expressed as a boolean function over the text of the test case, or over its syntax tree. Filters provide a simple way to reduce a test suite by keeping only the required tests. It is up to the engineer to develop a good filter to not eliminate relevant tests. Filters are not supported by TSLT, but they can be expressed using selectors.

The selector mechanism is another mechanism of Tobias to reduce the size of a test suite. While a filter applies to a single test case, a selector is applied to the test suite and gives a subset of tests that satisfies a given criterion. For example, standard selectors offered by Tobias, perform random reduction over the test suite. They select randomly a specified number of tests from the test suite. Using TSLT syntax we are able to define selectors in test patterns.

In Fig. 3.14 we present an example of test pattern `PurseSchemaWithoutSelector` that is a sequence of two group calls `checkPinGroup` and `TransactionGroup1To6`. `checkPinGroup` performs an authentication using the `checkPin` operation using 7 values for its parameter. `TransactionGroup1To6` group performs 1 to 6 calls to `credit` or `debit` operations using 4 values for their parameters. `PurseSchemaWithoutSelector` is unfolded into 2 097 144 test cases = 7 (Unfolding of `checkPinGroup` group) * 299 592 (Unfolding of `TransactionGroup1To6` group). We redefine the test pattern in `PurseSchemaWithSelector` by introducing selector mechanism to reduce the number of tests.

We specify the selector in TSLT using the `selector` key. For example, the selector `randomSelection100` is a Java random selector, it selects randomly 100 tests from a group unfolding. Its code is defined in the `SelectorRandom` Java class. To associate a selector to a group we use the `selectorgroup` key. For example `randomSelection_TransactionGroup1To6` is a selector applied to the group `TransactionGroup1To6`. It is called instead of the group call in the `PurseSchemaWithSelector` schema to unfold 100 tests from the unfolding of *TransactionGroup1To6*. The test pattern `PurseSchemaWithSelector` is unfolded into 700 test cases = 7 * 100 (result of Random selection from `TransactionGroup1To6` group unfolding). More complex selectors can be defined in Java by the Tobias user to exploit the code of the test cases, or

```
group PurseSchemaWithoutSelector {
  @checkPinGroup;
  @TransactionGroup1To6;
}

group checkPinGroup {
  iut.checkPin ([1234,5678,9123,4567,8912,3456,7891]);
}
group TransactionGroup1To6 {
  (iut.credit ([0,-1,100,5000]) | iut.debit([50,150,5000,-1]))
  {1,6};
}

group PurseSchemaWithSelector [us=true] {
  @checkPinGroup;
  @randomSelection_TransactionGroup1To6;
}
selector randomSelection100
  (int nb=100, int percent=-1, long seed=-1)
  [lang=java,file=SelectorRandom.class]

selectorgroup randomSelection_TransactionGroup1To6
  [groupid=TransactionGroup1To6,
  selectorid=randomSelection100, us=false]
```

Figure 3.14: A test pattern using selector technique

could even connect to the code or the specification of the system under test to measure some coverage.

### 3.4.5 Towards Model-based filtering

Combinatorial techniques allow exploring many different behaviors of the system by combining relevant input values. However, they do not rely on a specification of the SUT. Thus, generation may lead to a large number of tests that correspond to illegal inputs or sequences of calls whose execution results in inconclusive verdicts. This kind of test is called in the following *invalid* test.

A test which contains an operation call that violates the operation precondition is invalid. We take the example of *test 3*:

`iut.checkPin(1234); iut.debit(50);` presented in Fig. 3.5. We assume that before calling `checkPin` operation, the instance under test (iut) of the Purse class was created, initialized and the balance contains the zero value. The *test 3* is invalid if the precondition of `debit` states that the purse must store more money than the amount debited. For the test suite in Fig. 3.5, there are in total 26 invalid test cases.

The invalid tests can be useful for test robustness, which consists to execute the system with illegal inputs. However, in a context of conformance testing, one aims at checking whether the specification requirements are met or not in the SUT. In these conditions, invalid tests according to the specification must be discarded from the test suite.

We call a Model-based filtering strategy, the technique that relies on the use of models to filter out invalid tests. The idea is to execute or evaluate the tests against a specification. The next chapter presents research works proposing to use a specification to discard invalid tests and tests which do not provide the expected outputs.

Chapter 4

# Model-Based filtering of combinatorial tests

## Contents

In this chapter we present the notion of model-based filtering. This is our first contribution in combinatorial context to reduce a test suite. It consists in using a formal specification to filter out invalid test cases. We remind that an invalid test is a test that contains illegal inputs or sequences of calls whose execution (or animation) on the specification results in inconclusive verdicts. In this case, the specification provides the test oracle. It defines with unambiguous representation the requirements that have to be met by a system.

The specification of the SUT can be embedded in the system source code as JML specifications (Java Modeling Language). Using JML we are able to define system behavior aspects inside Java code. It can also be outside the source code in the modeling artifacts as OCL specification for UML designed systems.

Before presenting our filtering strategy (from Sect. 4.3 to Sect. 4.5), we first present in Sect. 4.1 and Sect. 4.2 the principle of the source code embedded specification and external specification. We give some examples of specification languages from the literature. We also present examples of researches that use these languages to evaluate test execution. In Sect. 4.6 we present the limits of our model-based filtering strategy. Sect. 4.8 gives some research works related to our contribution. Sect. 4.9 draws the conclusion of this chapter.

## 4.1 The source code embedded specification

Specifications can be embedded as assertion properties in the source code of the program under test, in a language extension to the programming language. These assertions are executed at run-time to check whether they are verified or not. These assertions are also called contracts and software development methods using such technique are called Design by Contract (DBC) methods [Meyer 1992].

Many embedded specification languages are reported in the literature. We present here Anna and JML.

### 4.1.1 Anna specification language

Anna (Annotated Ada) is an assertion language extension to Ada that allows to specify the intended behavior of programs [Luckham 1987]. It is the

```
function IntSquareRoot (X: INTEGER) return INTEGER is
--| where
--| X >= 0,
--| return R:INTEGER => R*R <= X and (R+1)*(R+1) > X;
begin
...
end IntSquareRoot;
```

Figure 4.1: Anna code

*"primary ancestor of many of the more recent executable assertion languages"* [Baresi 2001].

The Anna specification is inserted as annotations (comments fields) within Ada programs. The Anna annotated program is called a "self-checking" program, because for each annotated code encountered, it is checked for correctness to the associated code.

The specifications are defined in Anna language as a set of constraints. The violation of an asserted property raises the predefined error ANNA_ERROR. In Fig. 4.1, we give an example of a square root function for natural numbers. The precondition of the function (X>=0) is specified after the *where* keywords. The *return* keyword specifies the postcondition (R*R <= X AND (R+1)*(R+1) > X).

The annotations are marked as comments in the Ada syntax where each line begins with `--|`. In [Hagar 1996], the authors use the Anna formal specification language as a test oracle to check the correctness of an avionic control system.

### 4.1.2 JML specification language

Java Modeling Language (JML) is another embedded specification language [Leavens 2006, Cheon 2002]. It defines system behavior inside Java code. The correct execution of a method can be specified using invariants, pre- and post-conditions.

An invariant assertion can be used to define conditions that hold in all states of the class instance. The pre- and the post-condition have to be verified respectively before and after the execution of the method. JML specifications are written in special annotation comments, which start with an at-sign (@). JML uses the *requires* keyword to specify the client's obligation (pre-condition), the *ensures* keyword to specify the implementor's obligation (post-condition) and the *invariant* keyword to define an invariant condition. To check the JML assertions at runtime, they are translated into Java instruc-

```
//@ requires X >= 0;
/*@ ensures \result*\result <= X && (\result+1)*(\result+1) > X
@*/
public static Integer IntSquareRoot(Integer X) {
/*...*/
}
```

Figure 4.2: JML specification example

tions and added to the code of the specified program.

In Fig. 4.2, we define the pre- and post- conditions for the method IntSquareRoot.

In next section we present specifications that are defined outside the source code.

## 4.2 The source code external specification languages

The previous specifications have to be inserted in the program to test and expressed in specification languages intended to be checked at run-time. Other specifications can be defined outside the source code of the program under test. They can be used to design a system by translating the informal requirements into formal ones. Unlike the previous specification languages, these ones are completely independent from the technologies used for the system implementation. However, like the embedded specifications, these external specifications can be used as a test oracle to evaluate tests execution. VDM, Z and OCL are examples of such languages.

### 4.2.1 VDM language

The Vienna Development Method (VDM) was originally developed at the IBM laboratories [Bjørner 1978]. It is a method for modeling computer-based systems, using formal specifications. The VDM Specification Language (VDM-SL) allows to specify a system using mathematical objects, like sets, sequences, maps, etc. It has an extended form, VDM++, used to specify object oriented systems with parallel and real-time behavior.

The VDM-SL language has an executable character and it is evaluated in VDM tools environment as well as in the Overture open source tool built on top of Eclipse platform. In Fig. 4.3, we specify the pre- and post- conditions for the method IntSquareRoot using the VDM-SL language.

```
IntSquareRoot (X: int) R: int
pre X >= 0
post R*R <= X && (R+1)*(R+1) > X
```

Figure 4.3: VDM specification example

The VDM specification can be used as test oracle to check the correctness of tests. In [Aichernig 1999], the authors propose an approach to test a black-box system using VDM specification as test oracle. In [Ledru 2004], the authors use VDM specification to check the correctness of test cases generated from a combinatorial unfolding.

### 4.2.2 Z language

Using the Z formal specification language [Spivey 1989], one can specify the intended behavior of a system using familiar mathematical objects: sets, bags, functions, integers, etc. It is a language independent from the programming languages and the implementations details. Using the Z language the specification is decomposed into small pieces called schemas. They are used to describe both static and dynamic aspects of a system. The static aspect describes the state space by defining a set of attributes and their types. Moreover, it includes the invariant conditions that must hold for every state transition. The dynamic aspects include the operations, the input/output relationships and the possible state changes.

In Fig. 4.4, we give an example of a Z schema, representing the IntSquareRoot operation. Mikk [Mikk 1995] proposes an approach to generate an exe-

*IntSquareRoot*

$$X?, R! : \mathbb{N}$$

---

$$X? >= 0$$
$$X? >= R! * R!$$
$$X? < (R! + 1) * (R! + 1)$$

Figure 4.4: Z specification example

cutable test oracle from a Z specification by constraining specifications to an executable subset that can be translated into C or C++ code. To be able to execute the result, the process has to transform all the infinite types to

finite ones. The iteration inside the predicate must be finite. The quantified expressions ranges have to be finite or transformable to a finite one. In [Coppit 2005], the authors propose an approach for revealing faults by generating assertions from formal specifications (including Z) and inserting it into the source code.

### 4.2.3 OCL language

The Object Constraint Language (OCL) [OMG 2012] is another formal specification language. It is used within object-oriented models, mostly within UML diagrams. It allows to describe additional constraints about the objects in the model that can not be expressed by the graphical modeling language. Such constraints are usually described in natural language, but it may result in ambiguities. Using OCL it is possible to specify these constraints with precise and unambiguous representation. The OCL constraints may be invariants that must hold for every state of the class instance, or preconditions and postconditions that check the transition from a pre-state to a post-state upon an operation call. The OCL language allows to:

- navigate within the object-oriented model,
- manipulate sets and sequences of objects by performing special operations,
- to build first order (logic) statements by using universal/existential quantifiers.

Let's consider a class Math [Packevičius 2007] in UML containing a method IntSquareRoot to compute the square root for integer numbers. The method signature is defined as follows: public int IntSquareRoot (x: int). It is expressed in OCL in Fig. 4.5.

```
context Math::IntSquareRoot (x: int) : int
pre: x >= 0
post: (result*result) <= x AND ((result+1)*(result+1) > x)
```

Figure 4.5: OCL specification example

In [Cheon 2010], the authors propose an approach for automating the test oracle in Java programs, to filter randomly selected test data and determining test results. They define constraints in OCL language and translate them into runtime assertions, written in the Aspect oriented extension for Java (AspectJ).

We have seen in Sect. 4.1 and Sect. 4.2 the different specification languages used to evaluate test execution. These specifications are useful to decide on test validity. We remind that our problematic point is to discard invalid tests and tests which do not provide the expected outputs. We rely on a specification (or model) to evaluate the validity of the generated tests. In next section, we present our model-based approach to filter tests.

## 4.3 Filtering combinatorial test suites

Given an executable specification defined in a language $X$ and animation engine for the language $X$, it is possible to animate some tests. We can construct a tool that couples the test generator and the animator engine. The output of the test generator tool is the input of the animator tool. Doing this presents two advantages:

1. It is possible to filter the invalid tests according to the specification.
2. It is possible to record outputs of the animation and compute expected outputs as oracle.

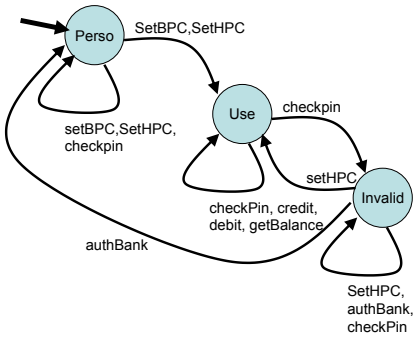
Our model-based approach is developed to resolve especially the problem of the test oracle that is not provided for Tobias generated tests. In our research work and in the context of the ANR TASCCC project, we use a UML/OCL specification as a test oracle. The Tobias generated tests are animated and filtered on an UML/OCL model using the CertifyIt tool developed by the Smartesting company[1]. In the following, we present a case study on which illustrations of our approach are presented.

### 4.3.1 Case study

We consider an example of a smart card application, representing an electronic purse (e-purse). This purse manages the balance of money stored in the purse, and two pin codes, one for the banker and one for the card holder.

The e-purse has a life cycle (Fig. 4.6), starting with a *Personalization* phase, in which the values of the banker and holder pin codes are set. Then a *Use* phase makes it possible to perform standard operations such as holder authentication (by checking his pin), crediting, debiting, etc. When the holder fails to authenticate three consecutive times, the card is *invalidated*. Unblocking the card is done by a banker's authentication. Three successive failures in the bank authentication attempts make the card return to the *Personalization*

[1] http://www.smartesting.com/




| Method signature | Informal description |
|---|---|
| `beginSession(int)` | Opening of session |
| `endSession()` | Termination of session |
| `setBpc(int)` | Sets the bank's pin |
| `setHpc(int)` | Sets the holder's pin |
| `checkPin(int)` | Identifies the holder |
| `authBank(int)` | Identifies the bank |
| `credit(int)` | credit of the purse |
| `debit(int)` | debit of the purse |
| `getBalance()` | value of the balance |

Figure 4.6: The main modes of the bank card and the main operations

phase. Each sequence of operations is performed within sessions, which are initiated through different terminals.

This example has originally been designed to illustrate access control mechanisms, and it is used as a basis for test generation for access control[2]. It was already used to illustrate test suite reduction with Tobias [Dadeau 2007]. The original example was specified in JML. We have translated this specification into a UML/OCL model for the Smartesting Test CertifyIt tool.

An example of the pre- and post-conditions of the `checkPin(int)` operation is given Fig. 4.7. The pre-condition requires that the session is opened using the `beginSession` method, the mode is put to *Use* by setting the user and the holder pin code, the terminal is *PDA* and the number of remaining attempts is positive. Post-conditions represent the code to be animated by CertifyIt if the pre-condition is verified. In the postcondition of the `checkPin` operation, if the pin code is correct, the card holder is authenticated else the number of allowed tries is decremented. If the number of maximum tries is reached, the card is set to the *Invalid* mode.

In next section we present the CertifyIt tool and its specific variant of OCL.

### 4.3.2 Automated oracle with the CertifyIt tool

The OCL supported by the CertifyIt tool is an imperative variant of OCL, inspired by the B language [Abrial 1996]. The variables appearing on the right hand side of `a = sign` are implicitly taken in their pre-state (usually denoted in OCL by `@pre`). In CertifyIt, information about the behavior of operations

[2]the original code of the application (in B and Java/JML) is available at
http://membres-liglab.imag.fr/haddad/exemple_site/index.html

**Method: checkPin(pin : int)**
**Pre-condition:**

```
(self.isOpenSess_ = true  and  self.mode_ = Mode::USE and
self.terminal_ = Terminal::PDA  and  self.hptry_ > 0) = true
```

**Post-condition:**

```
if (pin = self.hpc_) then /**@AIM: HOLDER_AUTHENTICATED */
  self.isHoldAuth_ = true  and  self.hptry_ = self.MAX_TRY
else /**@AIM: HOLDER_IS_NOT_AUTHENTICATED */
  self.hptry_ = self.hptry_@pre - 1 and   self.isHoldAuth_ = false and
  if (self.hptry_ = 0) then /**@AIM: MAX_NUMBER_OF_TRIES_REACHED */
    self.mode_ = Mode::INVALID
  else /**@AIM: MAX_NUMBER_OF_TRIES_IS_NOT_REACHED */
    true
  endif
endif
```

Figure 4.7: Pre and post-condition for `checkPin(int)` operation

is captured in assertions associated to the operations. In the perspective of animation, these assertions must characterize a deterministic behavior.

In fact, the OCL language is a *pure* specification language [OMG 2012]; it means that the evaluation of an OCL expression is done without effects on the model. It simply returns a value. The modified OCL version of CertifyIt allows to change the state of the instance on which the expression is called. The animation of an operation call on the model using OCL allows to change the values of the instance attributes when the expression contains assignments of values to these attributes. The initial state of an operation call in the sequence is the final state from the previous operation call animation.

Another construct of CertifyIt that can also be used in the operation OCL code is the tag clause. It is defined as a comment very often located in the conditional branches (for example in Fig. 4.7, the tags begin by the */**AIM* keyword). It is not evaluated by the animator but collected and displayed by the tool after each operation call animation. A set of tags covered by an operation animation represents an operation behavior. For instance for the *checkPin* operation, three behaviors can be identified:

B1 = {HOLDER_AUTHENTICATED}

B2 = {HOLDER_IS_NOT_AUTHENTICATED, MAX_NUMBER_OF_TRIES_REACHED}

B3 = {HOLDER_IS_NOT_AUTHENTICATED, MAX_NUMBER_OF_TRIES_Is_Not_REACHED}

The CertifyIt tool, is developed by the Smartesting company. It is used in

the context of the TASCCC project to provide an engine for test animation. The animation engine of CertifyIt takes as entries a test case as a sequence of operation calls and a UML/OCL model. The model contains the specification of the classes that represent the system under test. It contains also the class instances that will be used by the tests. The goal of these instances is to specify the initial state of the system for test animation. The goal of the test animations is :

- To verify the consistency of the specification
- To compute and observe the result (output and state) of the test
- To check the validity of the test according to the model

The answer of validity can be given by the tool after each operation call (called step) of the test case. If an instantiated call in the sequence violates the pre- or the post-condition of the corresponding operation, the animation stops and the test is considered as failed (invalid according to the model). CertifyIt also reports the tags covered by the last call and allows the evaluation of OCL predicates at any intermediate state in the sequence of operations.

In the next section, we present technical details about the process of test schema unfolding using Tobias and test animation using the CertifyIt tool.

### 4.3.3 Unfolding and animation process

The process of generation, animation and filtering of test cases by coupling Tobias and CertifyIt tools is presented in Fig. 4.8. The starting point is a schema file including a test pattern written in TSLT. Three steps are identified to produce the test evaluation results:

1. The schema file is unfolded by the Tobias tool which generates one or several test suite files written in the XML output language of the tool (outob file). For each group marked in TSLT as `us=true`, Tobias produces an outob file. This file contains all abstract test cases generated by the combinatorial unfolding of the corresponding group.

2. The outob files are translated into Java/JUnit test suites (using TDTest-Generator.xsl) including all necessary information to animate test cases. Each JUnit test case interacts with the CertifyIt API (TD API[3]) to be animated on the model (TD model file). We take advantage of the JUnit framework and the Java CertifyIt API to animate the tests in a popular and familiar tool for engineers, and to benefit from the JUnit structure of test suites.

[3]TD = Test Designer, a previous version of CertifyIt tool

3. JUnit executes the test suites. Each test case is animated on the CertifyIt model through the CertifyIt API. The animation process allows to identify and filter out invalid test cases, i.e. the ones which include an operation call that violates its pre- or post- condition.

Figure 4.8: The process of generation and filtering test cases (standard process)

The animation of test cases proceeds sequentially. If an operation call (step) fails, the animation of the test case stops and it is declared as failed and discarded from the test suite. The valid ones are saved to a test repository and used afterwards to test the application.

In the next section, we present an example of test schema defined in TSLT language. We also present the result of its Tobias unfolding and CertifyIt animation of the unfolded test cases.

### 4.3.4 Unfolding and animation process illustration

To illustrate the unfolding and animation process, let us consider the `EPurseSchema1` test pattern presented in Fig. 4.9. It begins by creating an instance under test. Next, it personalizes the card by setting the holder and banker pin codes. Then, it authenticates the card holder. Finally, it performs credit or debit operations using some amounts. The group that will be unfolded by Tobias is `EPurseSchema1` (`us=true`). It is a sequence of 4 groups: `IUT`, `Personalize`, `AuthenticateHolder` and `Transaction`. The IUT group defines a new instance of class EPurse. Then, the `Personalize` group opens a new `ADMIN` session, sets the banker and the holder PIN codes, and finally closes the session. The `AuthenticateHolder` group starts a session and checks the pin of the holder one to four times, and finally the `Transaction` group allows

```
group EPurseSchema1 [us=true, type=instruction] {
   @IUT;  @Personalize; @AuthenticateHolder; @Transaction;}
group IUT [type=instruction] { EPurse ep = new EPurse(); }
group Personalize [type=instruction] {
   ep.beginSession(Terminal.ADMIN);  ep.setBpc(@BankPinValue);
   ep.setHpc(@UserPinValue);  ep.endSession(); }
group AuthenticateHolder{
   ep.beginSession(Terminal.PDA); ep.checkPin(@UserPinValue){1,4}; }
group Transaction [type=instruction] {
    (ep.credit(@Amounts) | ep.debit(@Amounts)); }
group BankPinValue [type=value] {values = [12,45];}
group UserPinValue [type=value] {values = [56,89];}
group Amounts [type=value] { values = [-1,0,50]; }
```

Figure 4.9: A test pattern to illustrate the unfolding and animation process

to do credit or debit transactions. We use groups of values in some operation calls. For instance, the parameter of the `setBpc` method has 2 possible values defined in group BankPinValue: 12 and 45.

The `EPurseSchema1` pattern is unfolded into 720 test cases:
$IUT\ unfold = 1 *$
$Personalize\ unfold = (2 * 2) *$
$AuthenticateHolder\ unfold = (2^1 + 2^2 + 2^3 + 2^4) *$
$Transaction\ unfold = (3 * 2)$
$2^1$ in the AuthenticateHolder group unfolding corresponds to the unfolding of @UserPinValue (2 values) repeated one time. $2^2$ corresponds to the unfolding of @UserPinValue repeated two times, and so on for the 4 repetitions, and similarily for group transactions. In Fig. 4.10, examples of test cases unfolded from `EPurseSchema1` are given.

The abstract test cases produced by Tobias are translated to Junit tests and animated by CertifyIt tool on the electronic purse model. Only 168 test cases are reported by JUnit as succeeded. These are the valid tests (i.e. which satisfy the pre-conditions).

For instance, `TC3` is valid, contrary to `TC267` (which executes 4 consecutive calls to the `checkPin` operation with the wrong Pin code) and `TC720` (which executes a debit operation but never credits).

Let now consider that our objective is to make more combinations of credit and debit operations, by adding an iteration construct to `Transaction` group. For example, if we put an iteration `{1,10}` to the Transaction group, it would result into 8 707 129 200 test cases. This would be impossible to unfold because

```
...
TC3: EPurse ep = new EPurse(); ep.beginSession(Terminal.ADMIN);
     ep.setBpc(12);  ep.setHpc(56); ep.endSession();
     ep.beginSession(Terminal.PDA); ep.checkPin(56); ep.credit(50)
...
TC267: EPurse ep = new EPurse(); ep.beginSession(ADMIN);
       ep.setBpc(12); ep.setHpc(89); ep.endSession();
       ep.beginSession(Terminal.PDA); ep.checkPin(56); ep.checkPin(56);
       ep.checkPin(56); ep.checkPin(56); ep.credit(50)
...
TC720: EPurse ep = new EPurse(); ep.beginSession(Terminal.ADMIN);
     ep.setBpc(45);  ep.setHpc(89); ep.endSession();
     ep.beginSession(Terminal.PDA); ep.checkPin(89); ep.checkPin(89);
     ep.checkPin(89); ep.checkPin(89); ep.debit(50);
```

Figure 4.10: Examples of test cases unfolded from `EPurseSchema1`

Tobias would run out of disk space to store the resulting file. When a test pattern becomes complex by using many input values, iteration constructs and/or instructions set, it may correspond to a huge number of tests, that would be impossible to unfold. We call such test patterns as *explosive test patterns*.

In the section 4.4, we present new pattern constructs proposed to make it possible to take such explosive test patterns into account.

## 4.4 New pattern constructs

Here, we introduce three new constructs for the Tobias input language. These constructs support new techniques for filtering test cases. This allows to control the size of the produced test suite, and to incrementally pilot the combinatorial unfolding process. These constructs are inspired by the jSynoPSys scenario language [Dadeau 2009] and are syntactically and semantically adjusted to meet our needs.

### 4.4.1 The State predicate construct

The state predicate construct inserts an OCL predicate in the test sequence. The predicate expresses that a property is expected to hold at some point of the test sequence w.r.t. the model. Tests whose animations do not satisfy this OCL predicate at that point should be discarded from the test suite. It allows the tester to select a subset of the unfolded test suite featuring a given property at execution time.

For example, we can use this construct after *Transaction* group to select tests which result in positive balance. The pattern is defined as follows:

```
group EPurseSchema5 [us=true, type=instruction] {
  @IUT;
  @Personalize;
  @AuthenticateHolder;
  @Transaction⇝({ep} , self.balance_ > 0);
}
```

The TSLT construct takes the form ⇝`(set of targets , OCL predicate)`, where the set of targets identifies the objects (which correspond to `self` in the OCL predicate) on which the OCL predicate will be verified. Here, the set of targets is only one object that is the IUT `ep`. The OCL predicate requires that the balance value should be positive. The cases where we get a positive balance is when the credit operation is called with the value of 50. In the 168 valid tests unfolded from *EpurseSchema5*, only 56 tests result in positive balance. Therefore using this filtering construct more tests are discarded (112 tests were removed) comparing to the standard filtering (filtering tests which contain operation call that fails precondition).

### 4.4.2 The behaviors construct

We remind that the electronic purse specification is annotated by tags to distinguish the different behaviors of an operation. For example, the `checkPin` specification is annotated with tags in the conditional branches to differentiate a successful authentication from a failed one. When the tests are animated on the specifications using CertifyIt tool, it is possible to save the covered tags. It is then possible to filter tests on the basis of the covered tags.

We propose another filtering construct called behavior construct. It applies to an operation and keeps the tests whose animation covers a given behavior, expressed as a set of tags (see Sect. 4.3.2).

In the `AuthenticateHolder` group (Sect. 4.3.1), the tests that fail the authentication are valid tests because they verify the pre and post conditions of the *checkPin* operation. However, these tests do not allow to perform subsequent operations such as credit and debit. Therefore, we define a behavior construct to select the tests which succeed the authentication by selecting the authentication sequences whose last call to `checkPin` covers the tag `@AIM:HOLDER_AUTHENTICATED` (see Fig. 4.7). The test pattern is redefined as follows:

```
group EPurseSchema6 [us=true, type=instruction] {
  @IUT;
```

```
  @Personalize;
  @AuthenticateHolder2;
  @Transaction;
}
group AuthenticateHolder2 {
  ep.beginSession(Terminal.PDA);
  ep.checkPin(@UserPinValue){0,3};
  ep.checkPin(@UserPinValue)/w{set(@AIM:HOLDER_AUTHENTICATED)};
}
```

After the last call to `checkPin`, we put the symbol `/w` (with) and we define a set of tags that must be activated after the operation execution. Here, when the pin code is correct, the tag `@AIM:HOLDER_AUTHENTICATED` is covered in the post-condition of `checkPin` (see Sect. 4.3.1). When a test fails to cover the specified behavior by calling the checkPin operation, the animation is stopped and the test is declared as failed.

Using this construct at the point of authentication results in 168 valid tests that cover the `@AIM:HOLDER_AUTHENTICATED` tag. This is the same number of selected tests as for *EPurseSchema1* unfolding. The only difference between the two processes of unfolding and animation (using respectively *EPurseSchema1* and *EPurseSchema6*), is that using the behavior construct we avoid the animation of subsequent operations when the specified behavior is not covered. Therefore, by defining this construct we save time of unnecessary operation calls animation.

The predicate and the behavior constructs provide new ways to filter test cases. These new kinds of filtering are added in the third step of the process of generation and animation of test cases (see 4.3.3). In this step, in addition to the filtering according to operation pre- and post-condition, tests are discarded if they do not fulfill some state predicate or if they include some operation call that fails to activate the defined behaviors. Those constructs are used as directives in the test pattern to get the desired tests by discarding test cases which do not achieve the intent of the test engineer. The use of such filtering construct allows also filtering tests at early stage in the schema and avoid animation of failing subsequent operations (such as avoiding the animation of credit and debit operation in the example of EPurseSchema6 by using the behavior construct).

In next section we present another construct to filter test cases.

### 4.4.3 The Filtering key

The filtering key allows to select a subset of valid tests at some position and to discard the others. TSLT provides four filtering keys (`_ONE`, `_ALL`, `_n`,

_n%) to keep one, all, n or n% of the valid prologues. If we want to select all of them, we use _ALL. If we need just one, we use _ONE. _n (resp. _n%) randomly selects n (resp. n% of the) test cases amongst the valid ones. For example consider EPurseSchema7. The prologue group leads the purse to a state where the holder is authenticated. If the test engineer simply wants to keep one sequence which performs the prologue satisfactorily, he can add keyword _ONE after the prologue:

```
group EPurseSchema7 [us=true, type=instruction] {
  @Prologue_ONE;
  @Transactions;
}
group Prologue [us=true, type=instruction] {
  @IUT;
  @Personalize;
  @AuthenticateHolder2;
}
```

It means that first, the Prologue group will be unfolded. Second, unfolded elements are animated to get the valid ones. Third, **one** element is taken randomly from the valid ones and combined with the next instruction (@Transactions) to provide the final tests. More details about filtering keys processing of a test pattern similar to EPurseSchema7, will be given in Sect. 4.5.2.

The choice of the _ALL filtering key leads to safe filtering, since we keep all the valid tests at some specific point. However, the use of this key can be not relevant when the set of valid tests generated from the prologue is very large.

The filtering keys :_ONE, _n and _n%, can be used to select a smaller set of valid tests. However, they can omit relevant sequences for the subsequent operations.

By using filtering keys in the test pattern, we introduce a new concept of test pattern unfolding that consists in processing the test schema in many iterations. In each iteration, unfolding and animation techniques are performed for a sub-sequence of schema instructions. In the next section, we present the algorithm that allows to incrementally unfold and animate patterns by taking advantage of these filtering keys. We illustrate also by some examples, how it becomes possible to address explosive patterns using the incremental process.

```
1  algorithm Incremental_Generation_And_Execution_Process (p):
2   while( p contains at least one filtering key )
3     Let (prefix _1stKey ; postfix) match p in
4     validPrefixes := apply_Standard_Process(prefix);
5     validPrefixesSubset := Select_Subset_Of_
6                            According_To(1stKey, validPrefixes);
7     p := (validPrefixesSubset ; postfix);
8   end while
9   result := apply_Standard_Process(p);
10 end
```

Figure 4.11: The incremental unfolding algorithm

## 4.5 The incremental unfolding and animation process

### 4.5.1 Algorithm

The standard process of test unfolding and animation presented in Sect. 4.3.3 requires to completely unfold the test patterns and to animate each test case of each test suite. At this stage, we did not take advantage of filtering keys (`_ONE`, `_ALL`, `_n`, `_n%`). These filtering keys can be applied on the resulting test suite to select the relevant test cases. In this section, we will see that the early application of filtering keys may lead to significant optimizations of (a) the unfolding process and (b) the animation of the test suite.

The incremental process is defined for the unfolding of a single pattern `p`. It can be generalized to unfold multiple patterns. Its algorithm is given in Fig. 4.11 and performs the following steps:

- At each iteration, pattern p is divided into a prefix, located before the first filtering key, and a postfix, located after it (line 3 in the algorithm).
- The standard unfolding and filtering process of Sect. 4.3.3 is applied to the prefix. It results into a group of valid unfolded prefixes (line 4).
- A subset of this group is selected randomly according to the filtering key (see Sect. 4.4.3)(lines 5 and 6).
- This subset of valid unfolded prefixes is concatenated with the postfix to form the new value of p (line 7).
- The process iterates until all filtering keys are processed in the pattern (line 2).

- A last unfolding is applied to the resulting pattern stored in p (line 9).

### 4.5.2 Example

To illustrate this incremental process, we process the test schema `EPurseSchema1` by inserting the `_ONE` key after `AuthenticateHolder` group call. We get the following `EPurseSchema8` test pattern:

```
group EPurseSchema8 [us=true, type=instruction] {
  @IUT;
  @Personalize;
  @AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true)_ONE;
  @Transactions;
}
```

Before calling `@Transactions`, we would like to choose just one (`_ONE`) sequence of operations that succeeds holder authentication.
The prefix of this pattern is:

```
group EPurseSchema8pre [us=true, type=instruction] {
  @IUT;
  @Personalize;
  @AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true);
}
```

This prefix is then unfolded using the standard process. The three steps are executed to generate, animate and filter test cases. It unfolds into 120 tests, where 56 are valid. A valid test is chosen randomly amongst them and inserted as a prefix in the new pattern:

```
group EPurseSchema8b [us=true, type=instruction] {
  (ep.beginSession(ADMIN) ;  ep.setBpc(45) ;  ep.setHpc(56) ;
  ep.endSession() ; ep.beginSession(PDA) ; ep.checkPin(89) ;
  ep.checkPin(56) ; ep.checkPin(56) ;);   @Transactions;
}
```

Since there is no remaining filtering key, the whole pattern will be unfolded to generate the final test cases. This unfolding leads to 6 test cases (that corresponds to the result of the Transaction group unfolding), where only 3 are valid. The final number of valid test cases may depend on the prefix that will be chosen randomly. These test cases will be animated to discard the invalid ones, and then produce the filtered test suite. This process is clearly optimized since only 126 (120 + 6) test cases were completely unfolded, instead of 720 (120 * 6) in the standard process. Consider now the pattern `EPurseSchema9` that uses the `_ALL` filtering key instead of `_ONE`:

```
group EPurseSchema9 [us=true, type=instruction] {
  @IUT;
```

```
  @Personalize;
  @AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true)_ALL;
  @Transactions;
}
```

The prefix of the pattern is the same as for `EPurseSchema8` (`EPurseSchema9pre`). The 112 valid test cases generated from this prefix are inserted as a prefix. The new pattern created is:

```
group EPurseSchema9b [us=true, type=instruction] {
  ((ep.beginSession(ADMIN) ;  ep.setBpc(12) ;  ep.setHpc(56) ;
  ep.endSession() ; ep.beginSession(PDA) ; ep.checkPin(56);) | ;
  ...
  ((ep.beginSession(ADMIN) ;  ep.setBpc(45) ;  ep.setHpc(89) ;
  ep.endSession() ; ep.beginSession(PDA) ; ep.checkPin(89);
  ep.checkPin(89); ep.checkPin(89); ep.checkPin(89);));
  @Transactions
};
```

Unfolding EPurseSchema9b schema results in 672 (112 * 6) tests. Only 216 among them are valid tests. We can see the difference between using the filtering key `_ALL` and `_ONE` in terms of the number of tests generated. The advantage of using the `_ALL` key is that we produce all valid tests of the pattern. The more tests are produced from the schema, the more are likely to find errors and system vulnerabilities in the implementation. However, if the number of values and the number of operations in the operation group is large, the result of unfolding becomes large and it is thus more likely to suffer combinatorial explosion. Therefore, for a complex test scenario, the test engineer is the primary actor responsible to manage the test schema. He can see after which instruction the number of tests will be large, and then, he can insert the keys that he considers relevant at that point. If at some specific point, all the paths leading there are important, he uses then the `_ALL` key. Otherwise, he inserts the proportion key (n, n%, ONE) to select a subset of valid tests.

### 4.5.3 Potential adaptations of the approach

We have presented here a new approach introduced in the process of test schema unfolding. The standard combinatorial approach performs the values combination for all instructions in the test pattern. However, for big sets of values and/or big number of sets of values, the combinatorial unfolding will result in big number of tests that would be impossible to unfold or to animate/execute on the model/implementation.

The main idea of our approach is to process the test schema step by step.

A step in our context is a sequence of operation calls. We compute the result of unfolding for every step alone, we animate the resulted tests and we take only the valid ones. Taking the valid tests at early stages reduces the number of combinations, and then, helps to fight the combinatorial explosion.

The filtering mechanisms proposed in our approach are realized using constructs added to the test pattern language. The constructs proposed are specific to Tobias and CertifyIt tool, however they can be adapted to be used in other technical contexts.

The Tobias tool can be replaced by any combinatorial tool that makes possible to define a test schema, containing a sequence of operation calls applied with a set of values. The input language of the tool can be extended with the proposed filtering constructs: behavior and predicate filtering constructs. The filtering key is independent of the combinatorial tool. It is used only to incrementally unfold the test pattern and to avoid computing the combinations for the whole operation calls sequence.

The algorithm proposed in Fig. 4.11 processes the test pattern containing keys. Then, a test pattern containing a subset of instructions is produced that conforms to the input language syntax of the tool. The valid selected prefixes are inserted also in the syntax of the input language of the tool.

To perform the animation for the generated tests, the combinatorial tool is coupled with a model-based animator tool. The specifications used for filtering tests can be external to the source code as Z or B specifications, or internal to the source code as JML specifications.

For the case of JML, the tests are executed on Java implementation containing the JML specifications. The implementations are compiled using the *jmlc* compiler to translate the specifications into executable instructions, that can be checked at runtime. In the case of Z specification, the predicate construct has to be formulated in a Z syntax and verified using the Jaza tool. In the case of JML specification, the predicate is formulated as a Java assertion. This can be the case for other languages allowing the definition of assertions inside the code.

To perform behavior filtering, tags inside specification or source code must be defined to trace the covered operation branch. In Java/JML context, we can create a logging system that traces a tag inside every operation branches. The logging system generates a trace file containing for each test the list of tags covered. This file is then used to decide which test does not satisfy the behavior defined in the test pattern.

In the next section, we present the principal limitations of the approach, that have to be addressed in future works.

## 4.6 Approach limitations

The filtering keys used to incrementally process the test schema can only be used in the main schema that will be unfolded. The processing of other groups can not be addressed by our approach, i.e. we can not use the filtering key inside auxiliary groups to select a subset of valid tests from its unfolding.

Moreover, keys can not be used inside a disjunction of operations to select the valid prefixes for each of the operations inside the disjunction.

To illustrate this limitation, let us consider a group `mainSchema` that calls `Group1` representing a sequence of two operations (op0 and op1), then applies a disjunction (or choice) between operations (op2, op3, op4). The schema is described as follows:

```
group mainSchema [us=true, type=instruction] {
  @Group1;
  (op2(@V) | op3(@V) | op4(@V));
}
group Group1 {
  op0(@V);
  op1(@V);
}
group V [type=value] {values = [-1,0,1];}
```

A group of values $V$ is applied to the operation parameter. This disjunction represents the possible operation paths that can be traversed after calling the operation sequence `op0; op1`. For example `op0(-1); op1(-1); op2(-1)` and `op0(-1); op1(-1); op3(0);` are two possible unfoldings of `mainSchema`.

The key filtering may not be inserted inside `Group1`, e.g. after `op0(@V)` to take only the valid tests of the `op0` calls because pattern matching of Fig. 4.11 is not applied recursively. The key may only be inserted after calling `@Group1` to take the valid tests results from the sequence of calls `op0; op1`. If the number of calls corresponding to `Group1` is large, combinatorial explosion will take place before we can apply incremental filtering. The solution we use to resolve this problem is to copy and paste the definition of `Group1` into `mainSchema` and insert a key after the chosen operation call. The problem can also be addressed by changing the algorithm to match recursively the definition of groups.

Moreover, the key filtering can not be inserted inside the disjunction as for example inserting a `_ALL` key after `op2(@V)`. Once again, pattern matching will not work. It is only possible to insert the key after the whole disjunction, i.e. as follows:

```
(op2(@V) | op3(@V) | op4(@V))_ALL;
```

The insertion of keys after operations in the disjunction has the advantage to select a subset of valid operations paths at this stage and avoid computing all the possible combinations results from the whole disjunction group. For instance if it was possible to apply the `_ALL` key for `op2(@V)`, the algorithm will unfold only `op2(@V)` and not the whole disjunction. The algorithm then selects all (`_ALL`) succeeded prefixes from `op2(@v)` unfolding and insert it to replace the disjunction.

This problem can be resolved by changing the algorithm to process also keys inside disjunction elements. Only the elements that are followed by a key are unfolded. Prefixes are then selected from the valid ones according to the key. For instance, if we insert the `_ALL` key after `op2(@V)` and the `_ONE` key after `op3(@V)` then all valid prefixes from `op2(@V)` unfolding are selected and one valid prefix is selected from `op3(@V)` unfolding. The resulting valid prefixes taken from the two unfoldings are inserted in the schema to replace the disjunction.

Our approach allows selecting valid sequences of calls for a specific prefix. These valid sequences are inserted in the schema. Then, the test schema is unfolded and resulting tests are animated. The drawback of this technique is that the prefixes are animated before selecting them, and then reanimated after unfolding the new pattern. The animation of these prefixes is then performed twice and we know that their animation is valid. Therefore, there is an unnecessary time dedicated for reanimation. This problem can be resolved by memorizing the animation result after each succeeded sequence. In a new iteration, the results of animation of each succeeded sequence are taken by the animation engine as initial state used to launch the animation of the new operation sequences (not animated in the previous iteration).

In next section, we present improvements of our approach with respect to previous works developed for Tobias tool, to master combinatorial explosion using filtering mechanism.

## 4.7 Improvements with respect to previous works on Tobias

In [Ledru 2004, Ledru 2007], authors proposed two techniques to master combinatorial explosion with Tobias: test filtering at execution time, and test selection at generation time. Filtering at execution time is based on a simple idea: if the prefix of a test case fails, then all test cases sharing the same prefix will fail. In [Ledru 2004], an intelligent test driver is proposed which remembers the failed prefixes, and avoids to execute a test case starting with a prefix which previously failed. This idea is close to the one presented in

our approach. Still, there are significant advances in the new technique proposed here. First, the original technique required to produce the full test suite. Every test was examined to check if it included a failing prefix. Our new incremental process does not generate the full test suite, it incrementally builds and filters the prefixes by alternating between unfolding and animation activities. Because we avoid the full unfolding of the test suite, we are able to consider test patterns corresponding to huge numbers of test cases. Another advance of our approach is the definition of new constructs for test patterns (state predicates, behaviors, filtering keys), which help invalidate earlier the useless test cases in the unfolding process.

Selection at generation time is another technique, where one selects a subset of the test suite based on some criterion. This selection takes place during the unfolding process and does not require to execute or animate test cases. In [Ledru 2007] authors propose to filter the elements of the test suite whose text did not fulfill a given predicate. This predicate is freely chosen by the test engineer and does not prevent to filter out useful test cases. For example, one could filter out all test cases whose length was longer than a given threshold.

In [Dadeau 2007, Ledru 2007], authors investigated the use of random selection techniques. These techniques are by essence unable to distinguish between valid and invalid test cases, but they are able to reduce the number of test cases to an arbitrary number whatever be the size of the initial test suite.

Compared to these selection techniques, our incremental process does not discard valid test cases when using the `_ALL` key, but makes the assumption that the number of valid test cases is small enough to remain tractable. When `_ONE` or `_n` or `n%` is used random reduction takes place.

In the next section we present some related research works.

## 4.8 Related works

### 4.8.1 JSynoPSys

The research work the closest to ours is the one done by Dadeau *et al.* [Dadeau 2009]. They propose an approach to couple scenario based testing and symbolic model animation, implemented in a tool called JSynoPSys. Inspired from Tobias tool, Dadeau *et al.* propose to create scenarios in an expressive language used to generate test cases. Many constructs have been proposed in this scenario language, especially the ones that provide directives for test generation. They consist in restricting some behaviors (behavior construct) or properties (state predicate construct) in the resulting test cases.

Inspired from these constructs, we proposed the filtering mechanisms in TSLT to master the combinatorial explosion.

Unlike Tobias tool, the JSynoPSys tool avoids the enumeration of parameters values in the scenario. It describes only the succession of operations that a test should have, possibly with the intermediate states that should be covered by a test. The scenario description language of JSynoPSys tool contains three layers:

- the sequence layer, where regular expression are used to define the sequence of operation calls. For example, iteration or disjunction constructs can be applied on operation calls.
- the model layer, where operation calls at specification level are described (operations names), it represents the interface between the specification and the scenario.
- the directive layer, it makes it possible to use constructs to drive the test generation such as using the operation behavior coverage construct.

After defining a scenario, the tool instantiates the abstract parameters and generates test cases by performing the animation of a B formal model using the constraint solver of BZ-Testing-Tools. The animation is performed as follows. The abstract parameter is replaced by a symbolic variable handled by the constraint solver. Every operation is decomposed into behaviors. A behavior is defined by two elements: the predicate that indicates its activation condition and the substitution that represents the modification of the state variables. Then, the symbolic animation is performed by exploring the behaviors of operations defined in the scenario. When two operations are chained in the scenario, using backtracking mechanisms, the constraint solver enumerates all the possible combinations of behaviors for each operation. Instantiating the symbolic variables is performed by solving the constraints such as the variable input domain constraint, the system invariants and the operation pre- and post- condition. An operation sequence in a scenario is said as feasible if there exists at least one solution (by assigning values to the variables) after solving the related constraints.

To illustrate the JSynoPSys principle, let us consider an electronic purse application named Demoney [Dadeau 2009] defined in B language, similar to the EPurse application presented in Sect. 4.3.1. We used two commands (operations) of the system to create a scenario:

- `PUT_DATA(p, data)`: it personalizes a smart card by setting different card parameters (such as the maximum balance, the maximum flow and PINs values). In Fig 4.12, we give the B specification of the `PUT_DATA()` operation.

- STORE_DATA(): it validates the card personalization and puts the system in the use state.

```
Out ← PUT_DATA (p, data) ^=
  PRE
    p ∈ -128..127 ∧ data ∈ -32768..32767
  THEN
    IF (card_status = perso) THEN
      IF p = SET_MAX_BALANCE ∧ data≥ 0 THEN
        max_balance := data || out := sw_Success
      ELSE
        IF p = SET_MAX_DEBIT ∧ data  ≥ 0 THEN
          max_debit := data || out := sw_Success
        ELSE
         ... /* remainder of the operation */
        END
      END
    ELSE
      Out := sw_Error_life_cycle
    END
 END
```

Figure 4.12: B Specification of PUT_DATA operation

Let us consider the following test scenario defined using these two operations (this example was presented in [Dadeau 2009]):
`PUT_DATA {4} . STORE_DATA ⇝ (card_status = use)`

It consists in finding solutions that begin by calling the operation PUT_DATA 4 times, next call STORE_DATA one time and put the card in the use phase. To perform the symbolic animation from this scenario, abstract operation parameters are replaced by symbolic variables handled by the constraint solver and the behaviors of the operation are animated. An example of behavior of the PUT_DATA operation is:

```
p ∈-128..127+∧ data ∈ -32768..32767 ∧ card_status = perso ∧
p = SET_MAX_BALANCE  ∧ data ≥  0
⇒  max_balance := data || out := sw_Success
```

The first part (before the right arrow) represents the activation condition and the second part (after the right arrow) represents the substitution. The symbolic animation is performed successively for operations by exploring the possible behaviors combinations. It results in the following symbolic test case:

```
PUT_DATA(SET_MAX_BALANCE, X1) . PUT_DATA(SET_MAX_DEBIT, X2) .
PUT_DATA(SET_HOLDER_PIN, X3) . PUT_DATA(SET_BANK_PIN, X4) . STORE_DATA()
```

Using the following constraints:
$X1 \in 0..32767, X2 \in 0..32767, X3 \in 0..9999, X4 \in 0..9999, X1 > X2, X3 \neq X4$

Finally, a simple labelling technique is applied by solving the constraints and instantiating the symbolic variables. A possible test case is:

```
PUT_DATA(SET_MAX_BALANCE, 1) . PUT_DATA(SET_MAX_DEBIT, 0) .
PUT_DATA(SET_HOLDER_PIN, 0) . PUT_DATA(SET_BANK_PIN, 0) .
STORE_DATA()
```

We can summarize that the JSynoPSys technique relies on the use of model to filter invalid solutions, similar to our approach. Moreover, we use similar filtering mechanisms in the scenario language to master the combinatorial explosion due to the large number of combinations. For example, JSynoPSys tool can specify in a scenario whether all solutions will be returned for a specific iteration or just one.

Unlike our approach that animates test cases generated from combination of input values in the test pattern, JSynoPSys approach avoids the enumeration of values in the test scenario. The values are instantiated after a symbolic animation of the operation sequence and a constraints system solving. The example presented in [Dadeau 2009] does not give large examples of test pattern to experiment the solution finding capability of JSynoPSys in huge search spaces. However, such approaches suffer from combinatorial explosion because of the very large space, where solutions have to be found. This results from combination of values in large input domains. We assume that using some explosive test patterns (where many input parameters are used) such as the ones created for ECinema or Global Platform case studies, the constraint solver of BZ-Testing-Tools may be unable to find solutions.

### 4.8.2 Other related works

In [Jagannath 2009], authors propose to study test reduction in the context of bounded-exhaustive testing, which could be described as a variation of combinatorial testing. It is a technique to test an implementation by trying all inputs within defined bounds. This technique is time-consuming since the number of inputs is often large. Three techniques are proposed to reduce test generation, execution time and result inspection time: Sparse Test Generation, Structural Test Merging and Oracle-based Test Clustering.

Sparse Test Generation allows reducing the time that the user has to wait after launching the evaluation until the testing tool finds a failure. It is based

on the idea that failing inputs are closely located in the generated sequence of inputs. It allows to avoid exhaustive generation of all inputs but rather chooses one input from a closely located group.

Structural Test Merging allows reducing the total time for test generation and execution. It allows producing a smaller number of large tests rather than a large number of small tests. This technique is performed by merging appropriate program elements. Oracle-based Test Clustering allows to group failing tests such that all tests in the same group are likely caused by the same fault. This technique allows reducing the inspection time. Similar to our approach, the approach proposed in bounded-exhaustive testing context allows to reduce the number of generated tests and the time of execution (or animation) and evaluation. However, unlike our approach it does not rely on a model to perform the reduction of tests. It uses clustering techniques to reduce the number of inputs, and merging techniques to group a set of smaller tests into a single test.

In [Grieskamp 2009], authors combine the t-way combinatorial approach with a model-based approach. It allows generating combinations of actions parameters in the specification (a labeled transition system or finite state machine). In this approach, a t-way coverage requirement is combined with the path exploration technique to generate more parameter combinations than needed for covering the paths in the model. The generation of combinations is based on SMT constraints solver. The constraints are a set of conditions that have to be met by the generated solution. For instance, the range of a parameter is a constraint that has to be taken into account by the solver.

The approach has to meet two goals for the test generation: the interaction coverage goal and the path coverage goal. For instance, the generation engine begins by the generation of tests that cover pairwise (2-way) interactions. If there still remains some paths in the model not covered by the generated tests, more combinations are selected from the 3-way interaction coverage. The process begin from the 2-way generation to get the minimal set of tests needed to cover the model paths. If the 2-way generation has accomplished the path coverage goal, generating more combinations is not necessary.

The approach is integrated in Spec Explorer [4] from Microsoft, a model-based tool that performs model exploration by symbolic execution of the model code.

By summarizing the approach based on interaction and path coverage, we can identify two main techniques used by the approach to produce a reduced

[4]http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745/

test suite. First, it begins by the minimum interaction strength to generate the minimum set of tests needed to cover the model paths. Second, it integrates the constraint solver that allows to eliminate the tests that do not verify the constraints. Satisfying the path coverage goal in this approach can be considered similar satisfying the behavior coverage in our approach. Moreover, discarding tests based on constraints in this approach is similar to discarding tests based on predicates in our approach.
The test generation in this approach is carried out using a transition system in the model, however, in our approach, test generation is performed according to a test pattern that represents the set of test cases to unfold. Another point of dissimilarity, this approach is performing the t-way combinations for a single method, however, in our approach, the combination is performed exhaustively for a single method (that uses a set of values for its parameter) and between two method calls.

In [Nguyen 2012], the authors propose an approach to combine Model-based techniques and combinatorial testing to generate test cases. The model-based strategy allows to generate sequences of actions from models. Combinatorial testing defines the input combinations for the generated sequences. The approach is based on 5 steps:

1. Path generation: consists in generating paths (as sequence of events) from the model according to a specific criterion e.g. transitions coverage.

2. Path to classification tree transformation: consists in transforming each path into a classification tree by dissecting its elements: event, parameter and domain.

3. Test combination generation: t-way combinations are applied for the input domains in the different classification trees.

4. Post-optimization: consists in reducing the combinations repeated between classification trees for paths shared events.

5. Test execution and path constraint refinement: paths with input combinations are translated into executable test cases (e.g. for JUnit).

The test suite reduction appears in this approach in the step 4 (Post-optimization) by removing test cases that have redundant combinations. An algorithm is developed, taking as an entry test combinations for all paths. The algorithm analyzes the shared events and the input combinations and deletes redundant test cases by keeping the t-way coverage criterion satisfied.

Similar to our approach, this approach is combining the combinatorial testing and the model-based testing. It uses the combinatorial testing to

create combinations for the input parameters and get relevant tests. However, contrary to ours, this approach is using models to generate sequence of operations and not to filter tests. The test reduction is performed by an algorithm that compares the t-way coverage among tests.

## 4.9 Conclusion

In this contribution, we address the problem of filtering a large combinatorial test suite with respect to a UML/OCL Model. The whole approach relies on three main steps. First the set of tests to generate has to be *defined* in terms of a test pattern. Second, this schema is *unfolded* using a combinatorial tool to produce abstract test cases that are thirdly *animated* within an automated test oracle tool. This animation allows to identify and remove invalid test cases.

The process of unfolding and filtering can be done incrementally so that potential combinatorial explosion can be mastered. Several new constructs have been proposed in the input language of the combinatorial tool to help the test engineer to express more precise test patterns and to filter out invalid test cases at early stages of the unfolding process. From a methodological point of view, this requires to augment the test pattern with state predicates, behavior selectors, and filtering keys, which keep the incremental process within acceptable bounds.

The approach is described in this chapter using the Tobias as the combinatorial tool and CertifyIt as the automated test oracle tool. The three main sub-contributions that can be applied on other technical context are:

1. Coupling a combinatorial tool to generate tests and an automated oracle to discard invalid tests

2. Adding new constructs in the input language of the combinatorial tool to perform new filtering features

3. Applying an algorithm to incrementally unfold and check invalid tests by taking advantage of the new constructs proposed

In the next chapter we present illustrations of our approach on some case studies.

Chapter 5

# Using Model-based filtering on some case studies

## Contents



## 5.1 Introduction

In this chapter, we present illustrations of the approach performed using test patterns specified with the proposed filtering constructs (see 4.4). The experimented test patterns are explosive patterns whose unfolding using the standard process is subject to combinatorial explosion. The illustrations show how it is possible to unfold such patterns incrementally by applying the incremental unfolding and animation process that takes advantage of the new constructs.

A first illustration is presented in Sect. 5.2, performed for test patterns defined on basis of the case study presented in the previous chapter: Electronic Purse application. A second illustration is presented in Sect.5.3, performed on basis of a second case study called ECinema, a web application that allows to buy tickets for cinema movies. We present also a problem found in our

incremental approach, that will be discussed in detail in next chapter with solutions proposed to address it.

## 5.2 Illustration on E-Purse case study

We remind that the specification of the E-Purse case study is defined using UML/OCL for CertifyIt tool, presented in 4.3.1.

### 5.2.1 First example

Let us consider the following example:

```
group EPurseExample [us=true, type=instruction] {
  @IUT;
  @Personalize;
  @AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true);
  @Transactions{4};
}
```

The `EPurseExample` test schema performs in sequence an instantiation of the IUT, a personalization of the card, an authentication of the card holder by defining a state predicate to discard tests that fails the authentication and finally crediting or debiting the purse 4 times. `EPurseExample` is unfolded into 155 520 test cases. By totally unfolding the test schema, we succeed to achieve steps 1 and 2 (translation into TSLT and production of an outob file, see Fig. 4.8). Unfortunately, the translation of the outob XML file into a JUnit file crashes due to a lack of memory (we used up to 1.5Gb of RAM). If this had succeeded, we presume that the compilation of the JUnit file would also crash. These technical problems can be overcome by decomposing our files into smaller ones, but still the whole process would take time and computing resources. Other group definitions can rapidly reach over 1 million test cases which may require untractable time and memory resources.

Therefore, to make it possible to unfold the test pattern, we redefine it by introducing filtering keys:

```
group EPurseExampleUsingKeys [us=true, type=instruction] {
  @IUT;
  @Personalize;
  @AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true);
  @Transactions_ALL;
  @Transactions_ALL;
  @Transactions_ALL;
```

```
  @Transactions;
}
```

This pattern will produce the same valid test cases as the previous one, since we used the `_ALL` key. Using the incremental process, we need four iterations to remove the three filtering keys and unfold the resulting pattern. The pattern is completely unfolded and animated in 175 seconds as given in Fig. 5.1.

| Iteration | Nb of tests unfolded | Nb of tests accepted |
|---|---|---|
| 1 | 720 | 168 |
| 2 | 1008 | 560 |
| 3 | 3360 | 1904 |
| 4 | 11424 | 6496 |

Figure 5.1: Results of `EPurseExampleUsingKeys` unfolding

As a result our 155 520 test cases only include 6496 valid ones. To identify these, our incremental process needs four iterations but only unfolds and plays 16512 test cases. In this case, it performed the selection process using 10% of the resources needed for the standard one, and kept the test suites small enough to avoid tool crashes.

### 5.2.2 Second Example

Let us consider another explosive pattern, based on Fig. 4.6 called `EPurseSchema18op`. The aim of this pattern is to find test sequences where the purse goes back to Personalization mode, before being set in Use mode. The only way to reach this goal is to start from *Perso* mode, go into *Use* and *Invalid* modes, before getting back to *Perso* and finally to *Use*. These major steps are captured in the state predicates of the following pattern:

```
group EPurseSchema18op [us=true, type=instruction] {
  @IUT;
  @ALLOps{4}⇝({ep}, self.mode_ = Mode::USE);
  @ALLOps{5}⇝({ep}, self.mode_ = Mode::INVALID);
  @ALLOps{5}⇝({ep}, self.mode_ = Mode::PERSO);
  @ALLOps{4}⇝({ep}, self.mode_ = Mode::USE);
}


group ALLOps {
  ep.beginSession(@TerminalValue) | ep.endSession() |
```

```
  ep.setBpc(@BankPinValue) | ep.setHpc(@UserPinValue) |
  ep.authBank(@BankPinValue) | ep.checkPin(@UserPinValue) |
  ep.credit(@Amounts) | ep.debit(@Amounts);
}
group TerminalValue [type=value] { values = [ADMIN,BANK,PDA,NONE]; }
```

Group `ALLOps` contains all operations offered by the card possibly using a set of values for their parameters. It is unfolded in 19 elements. `EPurseSchema18op` repeats all operations 4 times, until it reaches the *Use* mode. Finding that it requires 4 iterations can result from a trial and error process, or from a careful study of the specification. The engineer has attempted to reach the *Use* mode in one to three steps, without success, and finally found that four steps were sufficient (session opening, setting the Holder and Bank codes, and session close). Similarly he found that 5 steps are the minimum to reach state *Invalid* (session opening, three unsuccessful attempts to checkPin and session close), and to then reach state *Perso* (session opening, three unsuccessful attempts to `authBank` and session close). We call this approach as *brute force* approach.

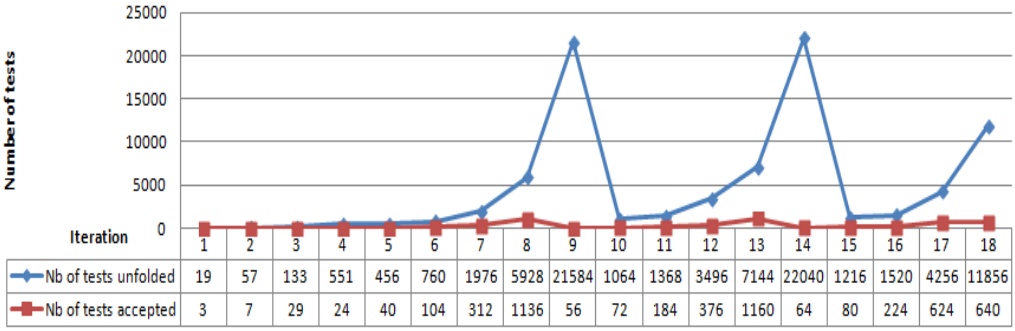
As a result, to find a valid sequence reaching the *Use* mode and returning to the same mode after visiting the other modes, we need to call at least 18 operations (4+5+5+4).

`EPurseSchema18op` represents $19^{18}$ test cases (about $10^{23}$ test cases), and thus cannot be directly unfolded. Because of the brute force approach, and because we inserted filtering predicates, a large number of these test cases will be invalid. This is typical situation where an incremental unfolding is relevant. To use it, we redefine `EPurseSchema18op` using the filtering key `_ALL` to restrict unfolding to valid prefixes.

```
group EPurseSchema18opWFilteringKey [us=true, type=instruction] {
  @IUT;
  @ALLOps_ALL;   @ALLOps_ALL;   @ALLOps_ALL;
  @ALLOps⇝({ep}, self.mode_ = Mode::USE)_ALL;
  @ALLOps_ALL;   @ALLOps_ALL;   @ALLOps_ALL;   @ALLOps_ALL;
  @ALLOps⇝({ep}, self.mode_ = Mode::INVALID)_ALL;
  @ALLOps_ALL;   @ALLOps_ALL;   @ALLOps_ALL;   @ALLOps_ALL;
  @ALLOps⇝({ep}, self.mode_ = Mode::PERSO)_ALL;
  @ALLOps_ALL;   @ALLOps_ALL;   @ALLOps_ALL;
  @ALLOps⇝({ep}, self.mode_ = Mode::USE);
}
```

`EPurseSchema18opWFilteringKey` is unfolded incrementally in 18 iterations. Fig. 5.2 shows the number of unfolded and accepted tests at each

iteration. It may be seen clearly that the number of unfolded tests increases




| Iteration | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Nb of tests unfolded | 19 | 57 | 133 | 551 | 456 | 760 | 1976 | 5928 | 21584 | 1064 | 1368 | 3496 | 7144 | 22040 | 1216 | 1520 | 4256 | 11856 |
| Nb of tests accepted | 3 | 7 | 29 | 24 | 40 | 104 | 312 | 1136 | 56 | 72 | 184 | 376 | 1160 | 64 | 80 | 224 | 624 | 640 |

Figure 5.2: Results of `EPurseSchema18opWFilteringKey` unfolding

with the growth of the accepted tests. This is because the number of unfolding of the *ALLOps* group is stable and equal to 19. This number is multiplied with the number of selected prefixes in iteration $i$ (accepted tests) to get the number of unfolded tests in iteration *i+1*. We can also see clearly in the graph that there are 2 peaks, one in the iteration 9 and the other in the iteration 14. This is because in their previous iteration (resp. 8 and 13), the numbers of accepted tests represent the two maximum values (resp. 1136 and 1160) of all accepted tests. The large numbers of generated tests of iteration 9 and 14 were addressed by using the filtering construct in the schema. In these steps, we can see how filtering predicates dramatically decrease the number of accepted tests. For instance from 21584 generated tests in step 9, only 54 tests were accepted. Fig. 5.2 shows that the number of test cases animated at each step remains small enough to be handled within reasonable time and computing resources, and to avoid tool crashes.

As a result, we unfolded and animated a total of 85 424 test cases for the 18 iterations in less than 17 minutes, instead of $19^{18}$ in the standard process. We finally found all 640 valid test cases hidden into this huge amount of potential test cases.

This second example shows that the incremental technique is efficient to find complex test cases hidden in a huge search space. The key to success is to make sure that the use of filtering keys will effectively reduce or limit the number of test cases at each iteration.

## 5.3 Illustration on ECinema case study

### 5.3.1 Specification of the case study

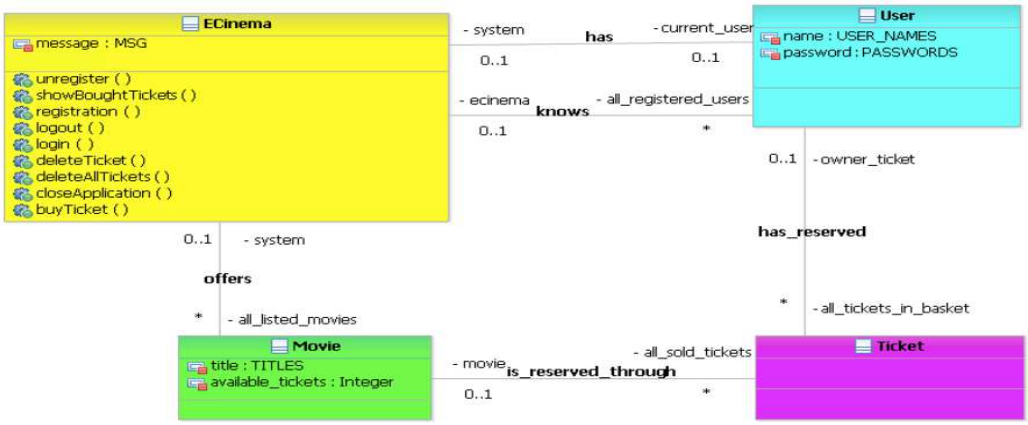



Figure 5.3: Class Diagram of the ECinema web application

ECinema is a web-application that allows registered and authenticated users to buy tickets for movies played in movie theaters [Dadeau 2013]. A list of available movies and their time sessions are displayed in the page. To be able to buy a ticket, the user must first be logged to the system. To log to the system, the user should be registered. To log in, the user must enter a valid user name and password. The valid user name is a registered user name. The valid password must match the password that corresponds to the valid user name. When logged in, the user can buy tickets.

The ECinema case study is used in the ANR TASCCC project to validate the complete chain of tools developed by project partners and resulting from this research project.

The UML class diagram is presented in Fig. 5.3. The ECinema class represents the class under test. The User class represents the registered and/or the connected users. The operations behavior is expressed using OCL specification, the OCL variant of CertifyIt tool. Fig. 5.4 gives the OCL code of the login operation that allows to authenticate the user by checking its login and password. The OCL code represents the post-condition of the operation. The precondition is always true.

It is reminded that code branches are annotated with special tags (starting with @REQ or @AIM) to trace requirements and identify specific behaviors

```
context login(in_name,in_password)::effect:
---@REQ: ACCOUNT_MNGT/LOG
if in_name = USER_NAMES::INVALID_USER then
 ---@AIM: LOG_Empty_User_Name
 message= MSG::EMPTY_USERNAME
else
 if not all_registered_users->exists(name=in_name) then
  ---@AIM: LOG_Invalid_User_Name
  message= MSG::UNKNOWN_USER_NAME_PASSWORD
 else
  let user_found : User = all_registered_users
  ->any(name = in_name) in
  if user_found.password = in_password then
   ---@AIM: LOG_Success
   self.current_user = user_found and
   message = MSG::WELCOME
  else
   ---@AIM: LOG_Invalid_Password
   message = MSG::WRONG_PASSWORD
  endif
 endif
endif
```

Figure 5.4: OCL code of the login operation of the ECinema system

of the operation. In the CertifyIt tool, it is possible to get the covered tags after a test animation.

### 5.3.2 Elements of illustration

In the context of the TASCCC project, the test patterns are not created manually by the test engineer. They are generated automatically from test properties [F. Dadeau 2013]. These properties express security requirements of the system using an ad hoc language. The test property language is a temporal extension of OCL, and describes with temporal patterns the correct execution of events sequences.

These test properties are then used either for computing the coverage of a property by executing a test suite, or for model-based test generation. The property language and its associated tools was defined by our collegues of Supelec and the University of Franche Comté. Here, we will not detail the

```
\**
Property 1. ‘‘Before logging on the system,
it is not possible to buy a ticket’’
*/
never isCalled(buyTicket(), {@AIM:BUY_Success})
before isCalled(login(), {@AIM:LOG_Success})
```

Figure 5.5: An example of a test property

language used for test property definition, the interested reader can refer to [Kanso 2013] or [Dadeau 2013] for more details. We present an example of test property defined on the ECinema case study and test patterns generated from this property.

We express in Fig. 5.5 a test property `Property 1` constraining `login` and `buyTicket` operations. It specifies that "before logging to the system it is not possible to buy a ticket".

From this property, different strategies exist to generate test scenarios (expressed in TSLT syntax). The strategies differ by the way they cover the property: nominal coverage tries to execute the transitions sequences accepted by the property. Robustness coverage strategy tries to execute transitions sequence not accepted by the property. Using the property `prop1` of Fig. 5.5, and by applying nominal strategies, the patterns generated are unfolded into almost 200 000 tests. However, using the robustness strategy we generate very explosive patterns.

We present in Fig. 5.6 an example of test patterns generated by robustness strategy. The main schema is the `sc_prop1_robustness` schema which is a sequence of group calls (`sequenceGroup0`). The `simpleOperationCall` groups define a set of operation calls using a set of values for their parameters. The `sequenceGroup` groups perform a sequence of group/operation calls. The `disjunctionGroup` groups define a choice between operations/groups (in this schema there is no disjunction groups). For example, `simpleOperationCall0` calls `setMinusGroup0` which defines a set difference between the set of all system operation calls (`base_call0`) and the elements of `call_restriction2` group. The complete test schema has 25 group definitions (it is presented in Appendix B). Therefore, because of the length of the generated test patterns, in the following, we only present the definition of the main sequence group (`sequenceGroup0`). The unfolding of the schema `sc_prop1_robustness` results in $1.89 * 10^9$ test cases, which is impossible to generate.

```
group sc_prop1_robustness [us=true] {
  @sequenceGroup0{1, 1};
}
group sequenceGroup0 {
@simpleOperationCall0{0,2};
@simpleOperationCall1;
@simpleOperationCall2{0,2};
@simpleOperationCall3;
}
group simpleOperationCall0 {
  (@setMinusGroup0)
}
group setMinusGroup0{
  SET = @base_call0 setMinus @call_restriction2
}
group call_restriction2 {
  ( @all_instances_ECinema.buyTicket(@default_enum_TITLES)
  | @all_instances_ECinema.login
  (@default_enum_USER_NAMES, @default_enum_PASSWORDS))
}
...
```

Figure 5.6: An example of a test schema generated from a test property in ECinema case study

### 5.3.3 Results of incremental process

Since the original schema `sc_prop1_robustness` can not be directly unfolded, we process the schema in two steps to make it possible to unfold. First, we replace the call to `sequenceGroup0` in the `sc_prop1_robustness` group by the sequence of operation calls defined in `sequenceGroup0` group. Second, we insert keys after each group call of the sequence to incrementally unfold the sequence. The new pattern created is presented in Fig. 5.7

We present in Fig. 5.8 the result of incremental unfolding of `sc_prop1_robustness_Keys` group. The incremental process is performed in 4 iterations in 154 seconds. The number of tests unfolded in total is equal to 29 148 tests = 2971 + 35 + 21246 + 4896.

```
group sc_prop1_robustness_Keys [us=true] {
  @simpleOperationCall0{0,2}_ALL;
  @simpleOperationCall1_ALL;
  @simpleOperationCall2{0,2}_ALL;
  @simpleOperationCall3;
}
...
```

Figure 5.7: A test schema generated from a test property processed with filtering keys

| Iteration | Nb of tests unfolded | Nb of tests accepted |
|---|---|---|
| 1 | 2971 | 7 |
| 2 | 35 | 6 |
| 3 | 21246 | 136 |
| 4 | 4896 | 76 |

Figure 5.8: Results of sc_prop1_robustness_Keys unfolding

### 5.3.4 Problems of explosive iteration

The maximum number of tests unfolded by processing `sc_prop1_robustness_Keys` test schema is 21 246 tests (see Fig. 5.8, iteration 3), and our tool is able to unfold and animate this number of tests. However, if the number of tests unfolded in an iteration becomes larger, e.g. 100 000 of tests, our tool crashes and is unable to give a final result for the schema unfolding.

```
group sc_1_prop3 [us=true] {
  //@sequenceGroup0{1, 1};
  @simpleOperationCall0{0,2}_ALL;
  @simpleOperationCall1_ALL;
  @disjunctionGroup0_ALL;
  @simpleOperationCall6_ALL;
  @disjunctionGroup1;
}
...
```

Figure 5.9: A test schema for which our approach fails to provide final valid tests

In Fig. 5.9, we present a test pattern `sc_1_prop3` generated from a test property `prop3` and processed by filtering keys. We insert the _ALL key after each instruction. The test pattern represents a sequence of operations and disjunction group calls. The incremental unfolding and animation approach succeeds to process the schema until the iteration 3. In this iteration, the number of tests to unfold is 1 658 423, that causes our tool to crash due a lack of memory (Java Heap Space). In fact, the number of elements unfolded from `disjunctionGroup0` is 127 571 and makes the iteration 3 explosive.

In next chapter we present the solutions proposed to deal with the problems of explosive iterations.

## 5.4 Conclusion

In this chapter, we presented some experimentations of our approach, performed on two case studies: EPurse and ECinema. In ECinema case study, we use some user-defined test patterns, which satisfy the intents of test engineer. The results show that our approach is able to find all valid tests in a huge search space ($19^{18}$ tests). In the ECinema case study, the experimented test patterns are generated from high-level test properties. It is shown that using the incremental unfolding and animation process, we get valid tests in a large search space ($1.89 * 10^9$ tests). We can conclude that using our approach, we are able to get valid tests from explosive test patterns created manually or generated automatically, that was impossible to unfold using the standard unfolding process.

By using our approach, it was also possible to see its limitation. The incremental unfolding process unfolds a test pattern in many iterations. However, when the number of tests unfolded in an iteration becomes very large, our process crashes and does not give final valid tests. In next chapter we present the solutions proposed for such problem.

Chapter 6

# Mastering explosive iterations

## Contents



## 6.1 Introduction

Using filtering keys, we are able to process explosive test patterns in many non explosive iterations. However, for some test patterns, the number of unfolded tests in an iteration can be very large and our tool crashes and is unable to give a final result. This problem is a limitation of our approach for which solutions have to be proposed. In this chapter, we give the solutions proposed to address problems of explosive iterations. We identified three cases that make an iteration explosive:

1. Explosive group unfolding (operation group and sequence group): In an iteration, a group may be unfolded in a very large number of elements making impossible to get valid prefixes in this iteration. In Sect. 6.2, we present the solution proposed to address problem of explosive group unfolding.

2. Explosive disjunction group unfolding: We remind that a disjunction group is a disjunction (choice) between sequence/operation/disjunction groups. When the number of input values used in the choices, or/and the number of possible choices (by using disjunction group inside disjunction/sequence group) is large, the called disjunction group in an iteration may be impossible to unfold. We propose different solution from the previous case. It is presented in Sect. 6.3.

3. Explosive instruction with repetition construct: some instructions are defined with the repetition construct (e.g. {0, 2}), it makes the number of elements unfolded in an iteration very large. In Sect. 6.4, we present the solution proposed to such problem.

The solutions proposed for each case are used to deal with the explosive test patterns generated from test properties in ECinema and Global Platform case studies.

## 6.2 Addressing explosive group unfolding

The number of tests unfolded in an iteration is equal to the number of accepted tests in the previous iteration multiplied by the number of calls unfolded from the group of the current iteration. For example, in the experimental result of Sect. 5.3.3 (Fig. 5.8), the number of tests unfolded in iteration 2 is equal to the number of tests accepted at the previous iteration (= 7) multiplied by the number of calls unfolded from `simpleOperationCall1` group (= 5). Therefore, to reduce the number of elements unfolded in an explosive iteration, we have either to reduce the number of valid prefixes selected in the previous iteration, or to reduce the number of unfolded elements in the group of the current iteration. We illustrate the problem by the following `sc1` test pattern. We suppose that the schema *sc1* is crashing at the second iteration, i.e. by unfolding the `seqGroup1` group.

```
group sc1 [us=true] {
  @simpleOpCall0_ALL; @seqGroup1_ALL; @simpleOpCall1;
}
group seqGroup1 {
```

```
  @simpleOpCall2; @seqGroup2;}

group seqGroup2 {
  @simpleOpCall3; @seqGroup3;}

group seqGroup3 {
  @simpleOpCall4; @simpleOpCall5;}
```

We apply the following two solutions to deal with this problem:

- Using proportion filtering key instead of the `_ALL` key (such as `_ONE`) in the **previous iteration** to reduce the number of selected prefixes (i.e. after `@simpleOpCall0` in our example). The `sc1` test pattern can be rewritten as follow:

  ```
  group sc1 [us=true] {
    @simpleOpCall0_ONE; @seqGroup1_ALL; @simpleOpCall1;
  }
  ```

- A second solution is proposed that can be used with or instead of the first solution. It consists in reducing the number of unfolded calls in the group call of the explosive iteration (i.e. `@seqGroup1`). This can be done by replacing the group call by its body instructions. The body instructions of the group are the instructions defined inside the group. By getting the body instructions instead of the explosive group call, we can insert keys after each instruction. The number of iterations increases and the number of unfolded tests in the processed iteration will decrease. This work can be done recursively to resolve the problem of combinatorial explosion in a specific iteration. This solution is only possible when the group is a sequence of instructions (operation and group calls). In the case of disjunction group, i.e. a choice between a set of group or operation calls, it is not possible to apply because our approach is unable to insert keys inside a disjunction (solution will be presented in Sect. 6.3).

By applying this second solution on our example, `@seqGroup1` group call will be replaced by its body instructions (`@simpleOpCall2; @seqGroup2;`), and the sequence groups can also be replaced recursively by their body instructions. The `sc1` test pattern can be then rewritten as follows:

```
group sc1 [us=true] {
  @simpleOpCall1_ALL; @simpleOpCall2_ALL;
  @simpleOpCall3_ALL; @simpleOpCall4_ALL;
  @simpleOpCall0;}
```

The `sc1` test pattern is now processed in 5 iterations instead of 3, and the number of calls unfolded in the iteration 2 is reduced from the unfolding size of `seqGroup1` to the unfolding size of `simpleOpCall2`.

## 6.3 Addressing explosive disjunction group

Unlike the sequence group, the disjunction group can not be replaced by its body definition. If we get a disjunction group in an explosive iteration, this is another problem. We propose two solutions to resolve this problem:

### 6.3.1 First solution

We begin by creating different schemas where each schema contains the instructions preceding the disjunction group followed by a call (group, operation) selected from the choices in the disjunction group. All the elements in the disjunction group choices are selected. The number of schemas created is the number of elements in the disjunction group choices. Fig. 6.1 illustrates the schemas created from the disjunction group (`disjGroup0_resol1` and `disjGroup0_resol2`). `_KEY` is either `_ALL`, `_ONE`, `_n` or `_n%`. Each schema created contains the prefix `@simpleOpCall0_KEY1` followed by an element from the disjunction group elements followed by the key defined after the disjunction group. The valid tests resulting from the unfolding of the created schemas, are intended to be inserted in `sc1` test pattern to replace the instructions sequence (`@simpleOpCall0_KEY1; @disjGroup0_KEY2;`).

- If KEY2 = `ALL`: it means that all valid tests from `disjGroup0_resol1` and `disjGroup0_resol2` unfoldings must be retained. We begin for example, by unfolding the `disjGroup0_resol1` to get the set of valid tests (denoted by Valid(`disjGroup0_resol1`)). Next, `disjGroup0_resol1` is unfolded and the valid test set result is collected (denoted by Valid(`disjGroup0_resol2`)). The total of valid tests Valid(`disjGroup0`) = Valid(`disjGroup0_resol1`) $\bigcup$ Valid(`disjGroup0_resol2`). Valid(`disjGroup0_resoli`) = $\emptyset$ if there are no valid tests (using **i**=1, 2). The valid tests collected from the union are inserted instead of the disjunction group and its preceding instructions as a disjunction of operation call sequences.

```
group sc1 [us=true] {
  @simpleOpCall0_KEY1; @disjGroup0_KEY2; @simpleOpCall1; }

group disjGroup0 {
  (@simpleOpCall2 | @simpleOpCall3) ;}

group disjGroup0_resol1 [us=true] {
  @simpleOpCall0_KEY1; @simpleOpCall2_KEY2;}

group disjGroup0_resol2 [us=true] {
  @simpleOpCall0_KEY1; @simpleOpCall3_KEY2;}
 ...
```

Figure 6.1: A second example of a pattern with explosive iteration

- If KEY2 = `n%`: we proceed as for the `_ALL` key to have all the selected tests from disjunction group (and its its preceding instructions) unfolding. Next, we insert the proportion (`n%`) of tests instead of the disjunction group and its preceding instructions as a disjunction of operation call sequences.

- If KEY2 = `ONE`: it means that one valid prefix (result from the unfolding of one of the created schemas) is sufficient. Therefore, we begin by unfolding one of the schema, if there are valid tests one test is taken randomly, else we try the next schema until having a valid test. The selected test is inserted instead of the disjunction group and its preceding instructions.

- If KEY2 = `n`: we begin by unfolding the first schema, the number of valid tests is computed. If we reach the intended number (n) in the current schema we stop the process, otherwise we continue the unfolding of another schema until getting the desired number of tests. The tests result are inserted instead of the disjunction group and its preceding instructions as a disjunction of operation call sequences.

### 6.3.2 Second solution

The implementation of the first solution requires to modify the algorithm of incremental unfolding, to take into account the processes described above for each key. A second solution is proposed, to resolve the problem of explosive disjunction group, without modifying the incremental unfolding algorithm. It

is used to unfold test patterns with explosive disjunction groups in ECinema and Global Platform case studies. We proposed a new key SUBSET that can be defined after a disjunction group call. The principle of this key is to select all valid prefixes results after calling one element from the choices in the disjunction group. If we get valid prefixes for this element we do not try another one. To select an element from the choices, the first priority is for the operation group, the second one is for disjunction group and the third priority is for sequence group. If we select a disjunction group, we have to select an element from its definition by following the priorities described previously. The selection of disjunction group makes the work to be performed recursively.

To illustrate this solution, in Fig. 6.2, we define the SUBSET key after the disjunction group. The created schemas (`sc1_resol1` and `sc1_resol2`) allow to resolve the pattern `sc1`. The `sc1_resol1` is created by choosing the element `simpleOpCall2` (priority 1). The `sc1_resol1` is created by choosing the element `simpleOpCall3` from `disjGroup1` (priority 2). Other schemas are also created by choosing the other elements. The `_ALL` key is following the element selected. We begin by unfolding `sc1_resol1`, if the problem of explosive iteration is resolved and we get valid tests at the final iteration, it is not necessary to try with the second schema. If the problem of explosive iteration is not resolved, or we do not get valid tests at the end, we try the next schema.

### 6.3.3 TestSchemaGen tool

The use of the disjunction groups inside disjunctions groups create different operations paths. Trying all the possibilities leads to several main schemas containing all possible operation paths. Therefore, to make easier the replacement task, we created a tool (TestSchemaGen) allowing to generate all possible paths from the original schema. Each path is generated as a test schema and represents a sequence of operation groups. The advantage of the tool is that it allows to generate in few seconds many possible schemas from the original schema. We add filtering keys to these schemas to fight the combinatorial explosion and find tests that satisfy the pattern. Moreover, if a processed schema does not provide valid tests we can try another one.

The advantage of this solution comparing to the previous one is that it takes less time to get the valid prefixes from unfolding a single schema. The drawback of this solution is that we do not get diverse valid operation calls by choosing one element from the choices comparing to the previous solution that can give diverse operation calls.

```
group sc1 [us=true] {
  @simpleOpCall0_KEY1; @disjGroup0_SUBSET; @simpleOpCall1; }

group disjGroup0 {
  (@simpleOpCall2 | @disjGroup1 | @seqGroup0) ;}

group disjGroup1 {
  (@simpleOpCall3 | @seqGroup1) ;}

group seqGroup0 {
  (@simpleOpCall4; @disjGroup2}

group sc1_resol1 [us=true] {
  @simpleOpCall0_KEY1; @simpleOpCall2_ALL; @simpleOpCall1;}

group sc1_resol2 [us=true] {
  @simpleOpCall0_KEY1; @simpleOpCall3_ALL; @simpleOpCall1;}
...
```

Figure 6.2: Explosive disjunction group and SUBSET key

## 6.4 Addressing explosive instruction using repetition construct

Another problem that can contribute to the combinatorial explosion is the iteration (repetition) construct. For instance, `@simpleOperationCall0{0,2}` of Fig. 5.6 represents a repetition of the group unfolding from 0 to 2 times. If we increase the upper bound to 4 for example, the number of tests increases from 2971 tests to 8 millions tests and will be impossible to unfold. Let us consider `@groupCall{m, n}` a group call repeated from m to n times making an iteration explosive. A solution to the problem can consist in reducing the number of repetitions of `groupCall`. First, we try to repeat it `(n-1)` times, if the iteration still explosive we try with `(n-2)`, and so on until reaching the `m` times. If `m` is equal to zero, it means that the call of the group is not mandatory and we can delete it from the schema.

Another solution consists in processing `@groupCall{m, n}` incrementally in k iteration where m <= k <= n. For example, `@groupCall{m, n}` can be processed as follows:

```
@groupCall_ALL;
```

```
...
@groupCall_ALL; //the kth element
```

In our work, the problem of explosive instruction using repetition construct is resolved by deleting the repetition construct. We choose to iterate the corresponding instruction at most one time. This solution can not be applied in some cases where the repetition of instruction is important to lead some intents of the test engineer (see Sect. 5.2.2), and therefore, the solutions proposed above are used instead.

## 6.5 Other solutions

We propose another solution that can be used to reduce the number of unfolded elements in an iteration. An operation group calls a set of operations with a set of values for their parameters. Instead of calling the operation group in the main schema we rather choose randomly an operation call from the set of operations offered in the corresponding group definition. This allows to reduce the number of combinations from the unfolding size of the group to the unfolding size of the operation chosen.

In some cases, unfolding an operation group in one iteration leads to millions of tests. This can be resolved as seen previously by choosing one operation call instead of the whole operation set. However, this technique can be ineffective because the chosen operation could invalidate the test, and thus we have to choose another operation.

Therefore, to resolve the problem of explosive iterations we introduce Tobias selectors in the TSLT schema (see Sect. 3.4.4). The Tobias selector selects a subset of tests from the tests that will be unfolded. We apply a random selector for the explosive schema group to select randomly a few number of operations from the millions of possible ones. The group call Xi to replace in the main schema is replaced with the random selector call to select tests from the unfolding of the group Xi.

In the case of the ECinema case study, the three problems presented above making an iteration explosive, are not common problems in the patterns generated from the test properties (defined in the context of ANR TASCCC project). We often replace just the sequence group in the main group by its body instruction. We insert then the keys (that can be in some cases the proportion keys) after each instruction and we run the process to get the final result without execution crash.

In the next section, we present illustrations done on the ECinema case study which contains the problems described above.

## 6.6 Some illustrations on ECinema case study

In ECinema case study, some test patterns generated from test properties are very explosive. Inserting keys after instructions in the main schema does not resolve the problem and the incremental unfolding results in explosive iterations. In the previous sections, we have proposed several solutions, depending on the case, to redefine the test schema to resolve the problem of explosive iterations. In this section, we apply these solutions to some test patterns used in the ECinema case study.

We used 3 test patterns: `sc_1_prop2`, `sc_1_prop3` and `sc_1_prop3bis` generated from different test properties. The test patterns are processed first by inserting filtering keys (`_ALL` key) after each instruction of the main schema. The incremental unfolding process crashes in some iteration due the large number of tests and fails to provide final valid tests. Therefore, we redefine the test patterns to resolve the problems of explosive iterations by applying the solutions described in the previous sections.

In Fig. 6.3, in the top we present the explosive patterns, the number of iteration ($NbI$) where the incremental unfolding crashes and the number of elements unfolded in this iteration ($NbT$). In the bottom we present the *resolved pattern* that represents the redefinition of the explosive pattern to resolve the explosion problems. $NMax$ gives the maximum number of tests unfolded in the incremental unfolding process. $TNb$ gives the total number of tests unfolded in all iterations. $NbA$ tells how many tests are accepted in final. $Nbs$ gives the number of seconds consumed to have the final result.

To resolve the explosive iterations, the disjunction groups are replaced by an element from the choices (disjunction group solution). If the choice is a sequence group, we take the sequence of group/operation calls. We can see in `sc_1_prop3bis` for example, how the disjunction group `disGroup0Prop3bis` is replaced by the body definition of a group sequence. Moreover, we apply the solution of using proportion key instead of the `_ALL` key to reduce the number of accepted tests in an iteration. For example, in `sc_1_prop3` after calling `simOpCall6Prop3` we select 4 valid prefixes instead of all valid ones. In addition to these solutions, we also use the solution of deleting the iteration construct in the first schema `sc_1_prop2`.

It can be seen clearly how the solutions proposed allow to resolve the explosive problems. By redefining the test patterns, it becomes possible to fight combinatorial explosion and to get final valid tests.

Our approach using the proposed solutions has been applied for 21 patterns generated from test properties and gives successful results for 12 patterns. The 9 other patterns are not giving valid tests in final. This can be explained by the fact that the test property created does not have valid tests satisfying it,

| Explosive pattern | | |
|---|---|---|
| Pattern | NbI | NbT |
| **1:** `group sc_1_prop2 { @simOpCall0Prop2{0,2}_ALL; @simOpCall1Prop2_ALL; @disGroup0Prop2; }` | 1 | 118 681 |
| **2:** `group sc_1_prop3 { @simOpCall0Prop3{0,2}_ALL; @simOpCall1Prop3_ALL; @disGroup0Prop3_ALL; @simOpCall6Prop3_ALL; @disGroup1Prop3; }` | 3 | 1 658 423 |
| **3:** `group sc_1_prop3bis { @simOpCall0Prop3bis{0,2}_ALL; @disGroup0Prop3bis_ALL; @simOpCall8Prop3bis_ALL; @disGroup2Prop3bis; }` | 2 | $3.4 * 10^{11}$ |

| Resolved pattern | | | | |
|---|---|---|---|---|
| Pattern | NMax | TNb | NbA | Nbs |
| **1:** `group sc_1_prop2 {@simOpCall0Prop2_ALL; @simOpCall1Prop2_ALL; @simOpCall2Prop2;}` | 11 115 | 15 851 | 702 | 54 |
| **2:** `group sc_1_prop3 {@simOpCall0Prop3{0,2}_ALL; @simOpCall1Prop3_ALL; @simOpCall2Prop3_ALL; @simOpCall6Prop3_3; @simOpCall13Prop3{0,2};}` | 10 623 | 15 390 | 48 | 92 |
| **3:** `group sc_1_prop3bis {@simOpCall0Prop3bis{0,2}_ALL; @simOpCall1Prop3bis_ALL; @simOpCall5Prop3bis_ALL; @simOpCall6Prop3bis_4; @simOpCall7Prop3bis{0,2}_ALL; @simOpCall8Prop3bis_ALL; @simOpCall15Prop3bis;}` | 21 224 | 47 780 | 840 | 331 |

Figure 6.3: Illustration of some explosive patterns on the ECinema case study

or because of the use of the SUBSET or the proportion key in the patterns to resolve the explosive iterations.

By using the SUBSET key, we launch the TestSchemaGen tool to generate all possible operation paths (from the disjunction groups) in different test schemas (see Sect. 6.3.3 for details). In our work we have not tried all the generated test schemas to find valid solutions. Therefore, we can not decide if the test property has or not valid tests.

Moreover, the use of proportion key may miss some relevant sequence that can make the test to be valid in final. Restricting the possible search space to find valid solutions, is actually a limitation of our approach.

It is interesting to note that the patterns used for the approach illustration are not defined by our team but rather generated from test properties defined by our project partners. We can also notice that the unfolding of the patterns is performed in reasonable time (few minutes or seconds), however the time of manual processing to resolve the problems encountered has to be considered (few minutes).

In next section, we present illustrations performed on Global Platform case study where very explosive test patterns are used.

## 6.7 Illustrations on Global Plateform case study

### 6.7.1 Description of the case study

Global Platform[1] is an industrial standard for the resource manager of multi-applications smart cards. It describes a set of features and interfaces for managing all aspects of card administration throughout its life cycle. Implementations of this standard are common, especially on credit cards to meet EMV (Eurocard-Mastercard-Visa) and the SIM and USIM cards for mobile phones, but also a great number of identity cards, electronic passports, health cards, etc. The features offered are:

- secure management of the life cycle of the card
- authentication of entities inside and outside the card
- secure communications with entities outside the card
- management of card contents, in particular applications
- routing of commands to the various applications

[1] http://www.globalplatform.org/

An important aspect of the Global Platform standard is that it is designed to allow multiple distinct actors (telephone operators, bank organization, transport operators, ...) to coexist on card.

In the ANR TASCCC project, the behavior model of Global Platform was designed by our partners using UML/OCL languages. The class diagram contains 84 classes. The class under test is **Card**, it contains 79 methods. This class presents 3 billions of possible atomic instantiated operation calls, due to combination of operations parameters values. Considering the complexity of the application, the properties defined by the test engineer to test the security aspects of this application will produce very explosive test scenarios which are unfolded to a huge number of test cases.

```
group sc_alpha3_temp1 [us=true] {
  @sequenceGroup0{1, 1};
}

group sequenceGroup0 {
@disjunctionGroup0;
@simpleOperationCall7;
@disjunctionGroup2;
@simpleOperationCall12;
@disjunctionGroup3;
}
...
```

Figure 6.4: An example of a test schema generated from a test property in Global Platform case study

### 6.7.2 Example of a test pattern

Similar to ECinema case study, the test patterns in Global Platform are generated from test properties written by the test engineer to specify an informal security requirement. We take an example of test pattern in Fig. 6.4 generated from a test property. The test property describes a constraint between two system operations: ExternalAuthenticate and InitializeUpdate. It specifies that ExternalAuthenticate operation call must be directly preceded by InitializeUpdate call. We used nominal coverage strategy for test pattern generation to produce transitions sequences accepted by the property (see Sect. 5.3.2 for details). The pattern generated is a representative example of explosive Global Platform test patterns. The test pattern has a size of 51 KB.

This test pattern contains 7 disjunction groups, 28 operation groups (group that contains a set of operations calls with a set of values for its parameters) and 9 sequence groups. It contains 32 different operations having from 0 to 12 parameters. The set of values for the parameters can reach 40 values. Combining the values of a simple operation call can generate millions of instantiated operation calls. Unfolding this pattern results in a number of tests $> 5 * 10^{100}$. This pattern contains explosive instructions, and thus we often use the proportion keys (`_n`, `_n%`, `_ONE`) as filtering keys to resolve explosive iterations and get valid tests at the end. The use of such filtering keys allows to reduce the number of unfolded tests in the next iteration and then master the combinatorial explosion.

### 6.7.3 Advantages of test pattern redefinition

To address the problem of explosive iterations, we process the test pattern similarly to the test patterns process in ECinema case study. We replace the disjunction groups and the sequence groups by the corresponding elements until having only the operation groups in the main schema. We remind that to replace an explosive disjunction group in the schema we select an element from its definition (choices). Performing this task manually is difficult and long because disjunction groups and sequence groups recursively contain many groups of the same type. For example, `disjunctionGroup3` has two elements: `disjunctionGroup4` and `sequenceGroup8`. Choosing `sequenceGroup8` for example, gives the following instructions `@disjunctionGroup7; @simpleOperationCall28;`, and so on.

Given the complexity of the test pattern by using disjunctions inside disjunction groups, we use the TestSchemaGen tool (see Sect. 6.3.3) to generate all possible paths (operation groups) in different test schemas. From the schema `sc_alpha3_temp1` of Fig. 6.4 we generate 36 different schemas representing all possible operation group sequences in the test pattern. In Fig. 6.5 we give an example of a schema generated from `sc_alpha3_temp1`.

Afterwards, we insert the `_ONE` key after each operation group, and we delete the repetitions `{0,2}`. This allows to minimize the tests unfolded in each iteration, however it can miss relevant valid prefixes.

In the iterations 3, 5 and 8, the number of tests generated exceeds one billion of tests. The corresponding operation groups make a call to one of the model operations (32 operations) using all possible values for their parameters except for 2 operations. Therefore, we replace the corresponding group call by a possible operation call (chosen randomly) rather than 30 operations to decrease the number of the combinations. The test pattern result is given in Fig. 6.6. The unfolding and animation process using this pattern is performed

```
group sc_alpha3_temp1 [us=true]{
  @simpleOperationCall0;
  @simpleOperationCall1;
  @simpleOperationCall2{0,2};
  @simpleOperationCall3;
  @simpleOperationCall5;
  @simpleOperationCall7;
  @simpleOperationCall8;
  @simpleOperationCall9{0,2};
  @simpleOperationCall10;
  @simpleOperationCall12;
  @simpleOperationCall13{0,2};
  @simpleOperationCall14;
}
```

Figure 6.5: Example of test schema generated from an original test schema

```
group sc_alpha3_temp1_keys [us=true] {
  @simpleOperationCall0_ONE;
  @simpleOperationCall1_ONE;
  @all_instances_Card.APDU_manageChannel
  (@default_enum_ALL_LOGICAL_CHANNELS,...)_ONE;
  @simpleOperationCall3_ONE;
  @all_instances_Card.APDU_manageChannel
  (@default_enum_ALL_LOGICAL_CHANNELS,...)_ONE;
  @simpleOperationCall7_ONE;
  @simpleOperationCall8_ONE;
  @all_instances_Card.APDU_manageChannel
  (@default_enum_ALL_LOGICAL_CHANNELS,...)_ONE;
  @simpleOperationCall10_ONE;
  @simpleOperationCall12_ONE;
  @simpleOperationCall13_ONE;
  @simpleOperationCall14;
}
```

Figure 6.6: Example of test schema generated from an original test schema processed with keys

in 12 iterations. The total number of tests unfolded is 2420. The small number is result to the test minimization techniques performed in the iterations (using

of the `_ONE` key, the repetition deletion and minimization of operation calls combinations). The whole process is executed in 28 minutes, and gives 1117 valid tests in final. The total number of accepted tests that satisfy the original test pattern is larger than 1117. This is due the test minimization techniques used to address explosive iterations. The process takes a long time for two reasons:

- The problem of the re-animation of the valid prefixes. This is noted as a drawback of our approach and we have given the solution to that in Sect. 4.6.
- The complexity of the OCL specification code of the operations used in test cases. The animation of an operation call with complex specification takes more time than an operation call with simpler specification. Moreover, in our illustration we are animating large test cases (number of operation call) and then, it takes long time to be animated.

In the context of Global Platform case study, we used very explosive test patterns. Therefore our objective is not to get all valid tests from the test pattern (that would be impossible), but get at least one test that satisfies the test pattern.

Actually, having 1117 tests for one property exceeds the number of tests that should need to be played on the smart card.

### 6.7.4 Combining filtering keys and Tobias selectors

To reduce the number of elements unfolded in an operation group, we choose only one operation call from the set of offered operations (as performed in `sc_alpha3_temp1_keys`). However, it is not guaranteed that by choosing this operation, the incremental unfolding process will give final valid tests. The chosen operation may make the generated tests invalid and thus we have to choose another operation call and re-execute the process to have valid tests.

Another solution that can be applied is the use of Tobias selector in test schemas. It allows to select few random elements from the explosive operation group unfolding. The advantage of using the random selection is that it is likely to get different operations and it is then likely to get diverse valid tests. The random selection is applied by replacing the explosive group call by the random selector call to select elements from the group unfolding. An example of selector definition is provided in Fig. 6.7. For example, the selector `randomSelection100` is a Java random selector, it selects randomly 100 tests from a group unfolding. To associate the selector to a group we define `randomSelection_simpleOperationCall2` that

```
group sc_alpha3_temp1_selector [us=true] {
  ...
  @randomSelection_simpleOperationCall2_ONE;
  ...
  @randomSelection_simpleOperationCall5_ONE;
  ...
  @randomSelection_simpleOperationCall9_ONE;
  ...
}
selector randomSelection100
(int nb=100, int percent=-1, long seed=-1)
  [lang=java,file=SelectorRandom.class]

selectorgroup randomSelection_simpleOperationCall2
  [groupid=simpleOperationCall2,
  selectorid=randomSelection100, us=false]
...
```

Figure 6.7: A test schema processed with selectors in Global Platform case study

is a selector applied to the group `simpleOperationCall2`. It is called instead of the group call in the main schema to unfold 100 tests from the unfolding of `simpleOperationCall2`. Using the same method, we replace `simpleOperationCall5` and `simpleOperationCall9` by the corresponding selector calls.

Unfolding the test pattern `sc_alpha3_temp1_selector` provides 1117 valid tests (same number of valid tests provided by `sc_alpha3_temp1_keys`).

## 6.8 Conclusion

We have presented illustrations and results of using our approach on several case studies and several test patterns. We have seen how our approach contributes to resolve the problem of explosive patterns written manually by the test engineer or generated from test properties. Using our approach, it becomes possible to find valid tests in a huge search space ($> 10^{100}$ tests). We also present the problems found when our approach is applied, especially the explosive iterations. We propose several techniques to deal with the problem, that we summarize in the following points:

- The use of the proportion keys (_ONE, _n, _n%) rather than the _ALL key. In this case, the objective was to find a subset of solutions that satisfy the test pattern rather than the complete set. The patterns processed by the proportion keys contains very explosive iterations and using the _ALL key will not resolve the problem.

- Replacing the groups with explosive unfolding by its body instructions to have its unfolding in more than one iteration. This process can be done recursively until having a non explosive iteration. We developed a tool to perform this process automatically and to have all the possible instructions sequences in different test schemas. The test schemas generated are the combinations of the elements that exist in the disjunction group body (instructions set).

- The use of Tobias random selector techniques to select a small number of tests from a huge number of tests in an iteration.

- For the instructions with the iteration construct, we choose to iterate it only one time.

- Choosing manually one operation from the instruction set to unfold it (choice or disjunction) to avoid unfolding all its elements.

The use of proportion keys in test patterns may avoid some interesting prefixes that make it possible to generate interesting valid tests. Moreover, the use of proportion keys restricts the possible search space. Then, in the case where no valid tests are generated from a test pattern, it is not possible to decide whether the test pattern has only invalid operation sequences or whether the solutions exist in the search space avoided by the proportion keys. It is more safe to use the _ALL all the time, and use the proportion keys when it is the only solution to perform. In some contexts, using the proportion keys can be prohibited, for example, when the security aspects defined in the test pattern are very important. In this case, getting all valid combinations from the test pattern is required. In other contexts, where it is suggested to generate some tests satisfying a defined test pattern. In this case, all valid combinations from a test pattern are not required.

The unfolding and animation process we have proposed is considered as a trials and errors process. Many varied attempts are performed by choosing different techniques and different inputs until having valid tests at the end. Multiple solutions proposed to fight the combinatorial explosion in an iteration are performed manually as replacing a group call in the schema by an element from its definition. However, these solutions can be integrated in our approach implementation to be performed automatically.

The case studies ECinema and Global Platform on which our approach are illustrated are given by a third party (our project partners). In these case studies, we have seen how it was necessary to us to propose a new filtering key (_SUBSET) that allows to accelerate the search of valid solutions.

Chapter 7

# Summary of the Model-based filtering contribution

## Contents



## 7.1 Motivation

Combinatorial testing allows to explore several system behaviors by combining relevant values. This technique is very efficient in practice to produce a large number of tests with minimum effort. However, this technique is often not relying on a specification to perform test generation. Therefore, as a result we get a large number of tests that correspond to illegal combinations of inputs or sequence of calls whose execution results in inconclusive verdicts. This is the case for example for tests which contain an operation call which fails the operation precondition.

## 7.2 The principle of model-based filtering

The solution we propose is to couple a combinatorial testing tool to an animation tool. The animation tool takes the tests generated from the combinatorial tool and animates them on a specification to decide which ones are valid and which ones are not. The tests that are *invalid* can be removed from the generated test suite. We call that technique model-based filtering in the way that it filters invalid tests based on a model (specification).

The implementation of our solution is performed using Tobias tool as a combinatorial testing tool and CertifyIt tool as an animation engine for the Tobias generated tests. The tests are generated from Tobias test patterns defined in the TSLT language. In a test pattern we combine different group of system operation calls using several values for their parameters. Other constructs can be also applied like the iteration construct. The Tobias generated tests are animated on UML/OCL model using CertifyIt tool to filter out invalid tests from the generated tests. The invalid tests are those which fail the OCL operation precondition.

## 7.3 Mastering combinatorial explosion

At this stage, we have resolved the problem of invalid tests in the generation stage by removing them. It is meaningful, since those invalid tests are not candidate to detect failures in system implementations, because they do not satisfy the specification.

However, complex test patterns (with many values, operations and/or iterations) can be impossible to unfold. Some patterns can be unfolded into billions of test cases. To be able to unfold explosive patterns, we propose an incremental unfolding process.

To perform the incremental unfolding process, we propose a filtering construct in the TSLT language called *filtering key*. The filtering key consists in a construct inserted after an instruction in the test schema in the form of _K (where K = ALL, ONE, n, n% or SUBSET). It introduces a new concept of test generation and animation process, that consists in incrementally processing the test schema. The incremental concept consists in unfolding the instructions before the filtering key. The tests generated from these instructions are animated to get the valid operations sequences. The valid results are inserted instead of the previous unfolded instructions. If K=ALL, we insert all the valid prefixes, if k=ONE, one prefix is chosen randomly from the valid tests and if K=n respectively n%, n or n% tests are chosen randomly from the valid tests. This technique avoids to unfold the whole test pattern in one unfolding and animation process, but rather unfold it incrementally in many iterations until dealing with all keys in the test schemas. In an iteration we take only the valid tests to be combined with the subsequent operations in the next iteration to avoid a large number of invalid combinations, and therefore we fight the combinatorial explosion.

In addition to the filtering key construct, we propose two other filtering constructs. The first one is the behavior filtering construct. It allows to filter out at some point in the test pattern instructions, the operation sequences that

do not cover a specific behavior of an operation. The second filtering construct is the predicate filtering construct. It allows to filter out at a specific stage in the test pattern, the sequence of operation calls that fails a specified OCL predicate. These two filtering constructs are kinds of directives that make possible to better target our desired tests. The filtering key can be coupled with the state predicate and behavior filtering constructs. In this case, in an iteration, the valid tests are verified according to the state predicate or to the behavior construct. Coupling the filtering key with a filtering construct (behavior or state predicate) allows to filter more test cases and to better target the desired tests in an iteration.

## 7.4 Results of using our approach in some case studies

By proposing the three filtering constructs we offer solutions to deal with the problem of explosive test patterns. We have experimented many test patterns on three different case studies: Electronic purse specification (EPurse), Electronic booking system of cinema ticket (ECinema) and Global Platform a last-generation operating system for smart card (GP). The electronic purse specification was constructed in our research team. We show using user-defined test patterns in the EPurse case study that our approach allows to find valid tests in a huge number of tests.

The ECinema and Global Platform case studies were provided by our research partners. The test patterns used for evaluation are generated automatically from test properties defined by the test engineer using an approach implemented by our project partners. The test patterns evaluated in ECinema case study can reach $9.3 * 10^{26}$ tests. Test patterns evaluated in GP case study are unfolded to more than $10^{100}$ tests. We have seen how our approach has resolved the explosion problem of many patterns.

Thanks to these case studies, it was possible to see the limit of our incremental unfolding approach. By inserting filtering keys in the explosive patterns, it was not directly possible to unfold the test pattern due the large number of tests generated in some iterations of the incremental unfolding process. Therefore, it requires the redefinition of the test pattern by reducing the number of elements generated in the explosive iterations and to make possible the incremental unfolding process to give final valid tests. The problem of explosive iteration is mainly due to explosive group call in the iteration. The main solution to this problem is to process the group call in the explosive iteration in extended iterations by processing incrementally its nested instructions in the explosive group definition. This task is performed manually and

recursively until having no explosive iterations. To help the test engineer to do that, a tool is developed to generate possible redefinitions of test schemas that may be proceeded instead of the original schema. Another solution is proposed to reduce the number of generated tests in an iteration: the use of Tobias selectors in test patterns such as the random selector, to choose randomly a subset of elements from the huge number of unfolded ones.

## 7.5 Conclusion and perspectives

Finally, we can conclude that our approach is used efficiently using many examples provided in intern and extern of our team. It can be adapted to be used on other combinatorial and animation tools, using other languages than TSLT for the test pattern and other specification languages than OCL.

The perspectives that concern this contribution consist in improving the test schema processing algorithm to take into account the tasks performed manually to resolve explosive iterations. Moreover, the current algorithm is reanimating prefixes already animated in previous iterations, therefore, an extension of algorithm has to be developed to memorize the results of animation of test schema prefixes.

# Test suite reduction using equivalence relations based on coverage of annotations inserted in source code/specification

Chapter 8

# Introduction

## Contents



## 8.1 Problematic

We remind that our thesis provides solutions to combinatorial testing issues. By using combinatorial testing, a large number of test cases generated from test patterns can be invalid according to the specification of the SUT. These test cases represent erroneous situations and do not fit into the set of test cases used to test the application. The first contribution (first part) of this thesis gives a solution to this problem by relying on a specification (model) to filter out invalid test cases at early stages. Using the Model-based filtering approach, the resulting test suite contains only the tests that are valid according to the model.

These valid tests are used to evaluate the implementation of the SUT. A few lines written in a test pattern can generate in few seconds a large number of valid tests. They are very useful to systematically observe the system behaviors and detect errors using many combinations of input values. However, in the context of regression testing, as many test cases are added to the test suite to evaluate new or modified requirements, the test suite grows rapidly and the cost of executing it becomes more expensive. This is because the translation of these test cases into a target technology such as JUnit, the compilation of the resulting file and its execution require too much computing resources.

Therefore, it is important to have a reduced test suite possibly to run it on the system. The question asked here is how to keep providing the

same capability of error detection with the reduced test suite. In general, this problem is defined as the test suite reduction problem [Harrold 1993]. Many researches have been investigated to propose approaches to resolve the problem and to provide a reduced test suite representative to the original one [Harrold 1993, Heimdahl 2007, Fraser 2007, Lin 2009, Parsa 2009]. For example, some approaches use the structural coverage information of test cases to reduce test suites [Harrold 1993].

In our research work, the problem of test reduction arises in a context other than regression testing context. We experimented embedded systems, especially to test security requirements of applications embedded in smart cards. The test engineer formalizes a security requirement as a test property defined in specific language [Castillos 2011]. Multiple test scenarios are generated from this property using approaches based on automata [F. Dadeau 2013]. From these test scenarios a large test suite is generated and filtered to cover all possible behaviors related to the security aspects defined before. Observing the evaluation of all these tests generated from the security property is interesting to detect errors and application vulnerabilities. However, in practice it is often impossible to evaluate all of them on the real application on the card due its limited resources. Therefore, we have to use a reduced test suite that is representative of the original one which can be executed on the card.

## 8.2 Solution

In this second part of thesis, we present a solution to this problem by proposing a new test suite reduction approach. The general idea of our test suite reduction approach is to take advantage of tests execution traces to define several similarity relations between tests. These relations are used to classify test cases and then to reduce the test suite by selecting a test case from each class.

Our work is inspired by approaches that use execution trace or coverage information to reduce a test suite [Jones 2001, Mcmaster 2005]. These approaches rely on standard coverage criteria as branch coverage or method coverage. Our approach differs from the existing approaches in two main points:

- It relies on coverage of annotations that can be inserted anywhere in the code for specific purpose and intended to be collected by test execution. Code annotation will be presented in Sect. 8.2.1.

- From the trace of annotations collected, a family of equivalence relations is proposed based on the order and the number of repetition of tags in

the test. These equivalence relations are used to compare the traces of tests and provide a reduced test suite. The equivalence relations and the test reduction system are presented in Sect. 8.2.2.

### 8.2.1 Code annotations

Code annotations, also called **tags**, can take several forms: from meaningful comments to calls to a special-purpose library. In our context, we only rely on the assumption that the annotations produce a trace when the associated code is executed. Executing a test case results in a trace of the covered annotations. Comparing the traces of two test cases provides raw material to decide on their equivalence.

The annotations can be used in various development contexts. It can be used for example to perform source code instrumentation as for debugging, profiling or code coverage measurement. The annotations can be also used for traceability purpose to trace back user requirements. For instance, the annotations inserted in the OCL specification of the model processed by CertifyIt fit into the requirement traceability context and each tag defined refers to a specific requirement. We note here that the idea of using covered annotations to reduce a test suite came from the use of annotations in CertifyIt OCL specifications.

In next section we present our test suite reduction approach.

### 8.2.2 Test suite reduction using equivalence relations

We assume the availability of a trace system that collects a tag when it is covered during test case execution. We remind that a test case is a sequence of operations calls. The execution of an operation call generates a sequence of tags. The test case execution generates a sequence of tags sequences.

Based on the sequences of tags sequences, a family of equivalence relations is proposed. These relations differ in the way they consider or not ordering or/and repetition of tags in the generated result. A rather strict equivalence relation requires that two equivalent test cases should feature exactly the same traces of tags. A more permissive equivalence relation simply requires the equality of the sets of tags present in both traces. This equivalence relation makes sense if the order of execution of the instructions of the test cases is not significant or relevant.

The test suite reduction algorithm takes as input the test suite to reduce, the execution traces of each test case and an equivalence relation. It compares the equivalence of tests according to the equivalence relation using their

traces. The equivalent tests are grouped in a cluster. The reduced test suite is produced by choosing randomly one test from each cluster.

### 8.2.3 Annotated specification

In the previous section, we present our approach applied in the context where we dispose of annotated source code and a trace system are available. Our approach can also be applied for annotated specification. In our work, we take advantage of CertifyIt functionality to collect tags to apply our approach. We insert tags in the OCL specification to trace system requirements. The CertifyIt tool collects the tags covered during animation of test cases. We then carry out the reduction using the covered tags and equivalence relation to provide a reduced test suite. The advantage of using our approach for an annotated specification, is to get reduced test suite without running it on the implementation of the SUT. This approach can be easily adapted to apply with other kinds of specification where annotations are inserted.

We experimented our approach on different case studies and examples of test suites generated combinatorially and randomly. The test reduction rate and fault detection capability are compared between the original and the reduced test suites. In these case studies, the annotations are inserted in source code or in the OCL specification to achieve requirement traceability or code coverage.

The organization of this part is as follow:
The chapter 9 presents the state of the art of test suite reduction approaches.
The chapter 10 presents our contribution with annotations inserted in specification or source code.
The chapter 11 presents preliminary experimentations and experimental results on a case study.

Chapter 9

# Test suite reduction

## Contents



## 9.1 Why reduction ?

One motivation behind performing test suite reduction is to reduce the amount of time and resources needed for recording or running a test suite. Deciding that a test suite is too large and needs to be reduced is relative. For example running ten thousands of test cases on a computer with 7 processor cores and 8 GO of memory resources can be an easy task. However, running the same test set on a smart card can be impossible due its limited memory resources and slow CPU. It would rather execute a representative subset of the test suite.

As well as the device capacity factor, the time factor can also be used to tell whether the test suite is too large or not. Ten thousands of test cases with the first computer configuration can be considered as large if the test running task has to be done in few minutes.

In the context of regression testing where software are continuously evolving, new test cases are added to the test suite to evaluate the new or the changed requirements. However, many test cases remain in the suite that could be either obsolete or redundant [Harrold 1993]. A test case is obsolete if it does no longer reference any functionality.

A test case is redundant if it provides the same coverage of program code (or the same coverage of requirements) than other test cases, for example with respect to a test criterion. Test suite reduction in the context of regression testing consists in deleting the obsolete and the redundant test cases.

The test suite reduction problem is widely addressed in the software testing literature and many approaches have been proposed resolving the problem.

We notice that a reduction in test suite may affect its fault detection capability. Many experiments have been investigated to see the effects of test minimization on the fault detection capability. The experiments show different results. Harrold, Jones [Jones 2001] and Rothermel et al. [Rothermel 1998] showed that the test reduction can dramatically decrease the fault detection capability. However, Wong et al. [Wong 1995] showed that the effect of test suite reduction on fault detection capability is not significant. To stay on the safe mode, since several studies have observed reduction in fault detection capability, we must be aware of this risk.

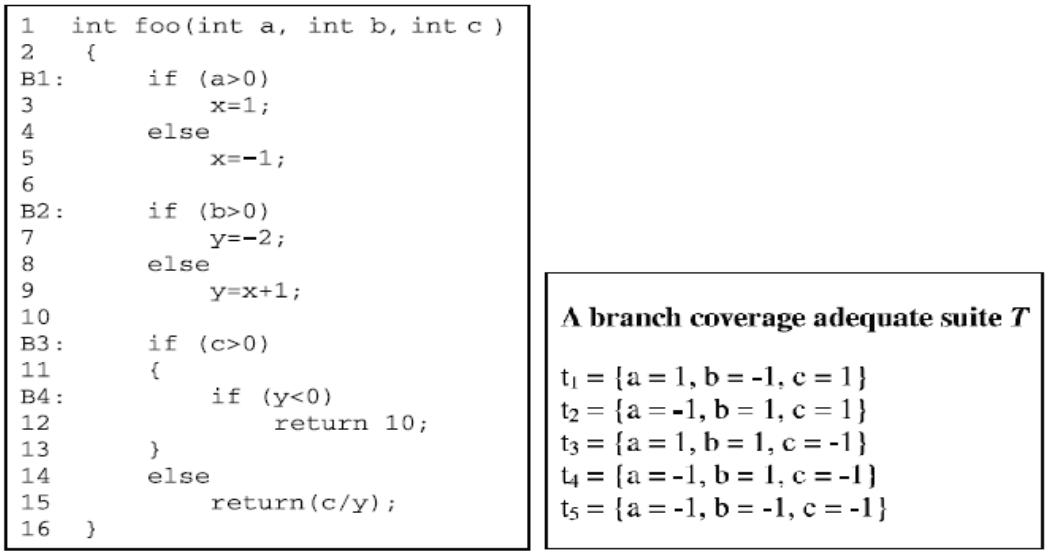
Two families of approaches reported in the literature performing this reduction: The *coverage-based* approaches and the *similarity-based* approaches.

## 9.2 Coverage-based test suite reduction

The coverage-based approaches take a representative set of test cases that still provide the same coverage of the program. They are based on the hypothesis that the higher is the coverage the more likely it is to detect faults. This was originally studied by Harrold *et al.* [Harrold 1993], and was later addressed by numerous authors [Lin 2009, Fraser 2007, Parsa 2009, Heimdahl 2007].

The coverage-based approaches uses coverage criteria to perform the test reduction. Multiple criteria have been proposed and explored [Zhu 1997]. We report some basic criteria which are the most used in literature:

- Control flow coverage criteria: These criteria are based on the coverage of the control flow graph of the program. The control flow graph represents all the paths that might be traversed by an execution of a program. For example:
  - Statement coverage: we consider a set of test cases is adequate if it covers every statement in the program at least once.



```
1   int foo(int a, int b, int c )
2   {
B1:     if (a>0)
3           x=1;
4       else
5           x=-1;
6
B2:     if (b>0)
7           y=-2;
8       else
9           y=x+1;
10
B3:     if (c>0)
11      {
B4:         if (y<0)
12              return 10;
13      }
14      else
15          return(c/y);
16  }
```

**A branch coverage adequate suite *T***

$t_1 = \{a = 1, b = -1, c = 1\}$
$t_2 = \{a = -1, b = 1, c = 1\}$
$t_3 = \{a = 1, b = 1, c = -1\}$
$t_4 = \{a = -1, b = 1, c = -1\}$
$t_5 = \{a = -1, b = -1, c = -1\}$

Figure 9.1: Simple program and test suite

  - Branch coverage: This type of coverage requires that every control transfer (such as IF statement) in the program under test is proceeded by at least one test case.
  - Path coverage: It requires that all possible paths of the program are executed by the tests.

- Data flow coverage criteria: These criteria are based on the coverage of the data used in the program. For instance:
  - All definitions criterion: It requires that all definitions occurrences of variables must be covered. For each definition occurrence covered the test must cover a path through which the definition reaches a use.
  - All use criterion: It requires that all uses of a definition should be covered.

- Fault-based adequacy criteria: It is based on the measurement of the fault detection capacity of the test set. This is based on mutation testing. The principle is to insert artificial faults in the program and verify if it is detected by the test. The program that contains the fault is called mutant. When we execute a test case, if a mutant produces a different result than the original program the fault is detected and we say that the mutant is killed. Otherwise, the mutant is alive. The percentage

| | $B_1^T$ | $B_1^F$ | $B_2^T$ | $B_2^F$ | $B_3^T$ | $B_3^F$ | $B_4^T$ | $B_4^F$ |
|---|---|---|---|---|---|---|---|---|
| $t_1$ | × | | | × | × | | | × |
| $t_2$ | | × | × | | × | | × | |
| $t_3$ | × | | × | | | × | | |
| $t_4$ | | × | × | | | × | | |
| $t_5$ | | × | | × | | × | | |

Figure 9.2: Branch Coverage of test suite

of killed mutants by the test set is called the mutation adequacy or the mutation score. Based on this measurement, we can define multiple fault-based criteria such as if the execution of the test set reaches a specific percentage of mutation score, it is then an adequate test set.

The problem of getting a representative set of test cases with respect to a testing coverage criterion is defined as follows [Harrold 1993]:

Given: a test suite TS created for a program P, a set of test case requirements R=$r_l$, $r_2$,..., $r_n$ that must be satisfied to provide the desired testing coverage of P.

Problem: Find TS' a representative set of test cases from TS, that satisfies all of the $r_i$ in R.

In this section we present 4 algorithms to build TS'. They share the same structure presented in [Sprenkle 2005]:

1. Initializing TS' as empty set.
2. Select a candidate test case t from T and add it to TS'.
3. Repeat 2 until TS' satisfies R.

We will illustrate the four algorithms using the example of Fig. 9.1. It presents a simple program and the corresponding test cases used to test it with respect to the branch coverage requirements. The program accepts three integers and returns a value. The example is defined in [Lin 2009].

Fig. 9.2 presents the branch coverage of the test suite.

### 9.2.1 Random reduction

A naive approach solves the problem by taking randomly [Sprenkle 2005] the test case in the step 2. The implementation of the random approach is the easiest one, however it does not provide an optimal reduction. The result of

reduction depends on whether or not the test case chosen in step 2 covers the maximum number of not yet covered requirements. In our example, the random approach can begin by selecting t5 and it is added to TS'. Next, in order t4, t2 and t1 are chosen. After choosing t1, all the requirements are satisfied and the algorithm is finished. The result is then TS'={t5,t4,t2,t1}. If the algorithm had selected t1 after choosing t5, more requirements would be satisfied comparing to t4. After choosing t1, if the algorithm selects t2 all the requirements will be covered by three tests. The result will be TS'={t5,t1,t2} which is better than the previous result. The disadvantage of the algorithm is that it might select a candidate, even though it does not provide more coverage than the current reduced test suite.

### 9.2.2 The greedy approach

Using the greedy algorithm [Chen 1998], the next test case to choose has to satisfy the maximum number of unsatisfied test requirements, i.e. it should provide the most coverage improvement.

On our example Fig. 9.1 and Fig. 9.2, greedy algorithm will first select either t1 or t2 because they have the maximum coverage of requirements. If it selects t1, next, it will select t2 or t4 because each of them covers more different requirements than the other tests. If the algorithm chooses t2, it will then select any of the remaining tests because all of them cover the requirement B3F that remains to satisfy. The result could be then TS' = {t1, t2, t5}. The result is better than the first solution produced by the random one, but it requires more time to compute which of the tests have the maximum coverage improvement.

It is clear that a greedy algorithm gives on average a better solution than the random approach, however the greedy's approach disadvantage is the time taken to compute the improvement in maximum coverage among tests.

### 9.2.3 The HGS algorithm

Harrold et al. propose an heuristic (HGS algorithm) to reduce a test set [Harrold 1993]. The heuristic begins by creating subsets from TS to associate each requirement with the set of test cases that satisfy it: T1, T2,... , Tn. Each $T_i$ contains the test cases that satisfy the requirement $r_i$. In Fig. 9.3 we present the subsets $T_i$'s for our example.

The algorithm first adds to the representative set all test cases in the $T_i$'s featuring a single element and marks all $T_i$'s containing these test cases. In our example Fig. 9.1 and Fig. 9.2, the algorithm adds t2 and t1 to TS' (because T7 and T8 have a single element) and marks T1, T2, T3, T4, T5,

| i | $r_i$ | $T_i$ |
|---|---|---|
| 1 | $B_1^T$ | {t1, t3} |
| 2 | $B_1^F$ | {t2, t4, t5} |
| 3 | $B_2^T$ | {t2, t3, t4} |
| 4 | $B_2^F$ | {t1, t5} |
| 5 | $B_3^T$ | {t1, t2} |
| 6 | $B_3^F$ | {t3, t4, t5} |
| 7 | $B_4^T$ | {t2} |
| 8 | $B_4^F$ | {t1} |

Figure 9.3: Association between requirements and set of test cases that satisfy it

T7 and T8. Then, the algorithm processes all unmarked $T_i$'s of cardinality two, and chooses the test case that occurs in the maximum number of $T_i$'s of cardinality two. In our example, all $T_i$'s with cardinality two are marked. The algorithm processes then the unmarked $T_i$'s with cardinality three and chooses the test case that occurs in the maximum number of $T_i$'s of cardinality three. This process is continuously repeated from 4 to max, where max represents the maximum cardinality of the $T_i$'s.

When examining the $T_i$'s to select the test case, several test cases can occur in the maximum number of $T_i$'s of the same size, and this case is called a tie. In the case of tie for $T_i$'s with cardinality n, the heuristic examines the unmarked $T_i$'s with cardinality (n+1) for the test cases that were involved in the tie. If it still exists a tie in the cardinality (n+1), greater cardinality is examined and finally the algorithm makes a random choice in the case where the max cardinality is reached. In our example, in the cardinality three, there is only one unmarked $T_i$ that is T6. Because 3 is the maximum cardinality the algorithm makes a random choice between t3, t4 and t5. The final result could be then {t1, t2, t3}, {t1, t2, t4} (or {t1, t2, t5} as in Sect. 9.2.2).

### 9.2.4 The GRE algorithm

The GRE algorithm presented by Chen and Lau [Lin 2009] considers two types of test cases: the essential test case and the 1-to-1 redundant test case. A test case is said to be essential if it is the only one that covers a specific requirement. A test case is regarded as 1-to-1 redundant if its covered requirements is a subset of covered requirements of another test case.

Three strategies are applied alternatively by the GRE heuristic until all requirements are satisfied: (1) the essentials strategy selects all essential test

cases, (2) the 1-to-1 redundancy strategy removes 1-to-1 redundant test cases, and (3) the greedy strategy selects test cases that meet the maximum number of unsatisfied requirements.

The selection of essential test cases might cause 1-to-1 redundant test cases. Removing 1-to-1 redundant test cases might generate more essential test cases. Therefore, the GRE algorithm applies alternatively these two strategies when possible. Otherwise the algorithm uses the greedy technique.

In our example Fig. 9.1 and Fig. 9.2, there are two essential test cases that will be selected: t1 and t2 (see Fig. 9.3, t1 is the only one that covers B4T and t2 is the only one tat covers B4F) After that, there is no 1-to-1 redundant test cases to remove, the algorithm will then apply the greedy strategy. It chooses the next test case that covers the most unsatisfied requirements (B3F), which will be one of the three remaining test cases. The result could be then TS' = {t1, t2, t3}, {t1, t2, t4} or {t1, t2, t5} as in the previous section.

A recent empirical study has been investigated to compare Greedy technique, HGS and GRE algorithms [Zhang 2011]. Several realistic test suites have been experimented on large programs. The authors evaluate the benefits and costs of these test-suite reduction techniques. They suggest to use the HGS algorithm in practice to achieve cost-effective reduction.

## 9.3 Similarity based test suite reduction

In contrast to the previous approaches that consider that test cases which have more coverage of the program have better fault finding capacity, the similarity based approaches consider the hypothesis that the more diverse test cases are the more likely to detect faults they are.

These approaches can rely on the use of a similarity function to measure test cases diversity. The similarity function can use information about the tests such as their code coverage. We present in next section two research works that study test reduction using similarity measures based on code coverage.

### 9.3.1 Similarity-based reduction using code coverage information

In [da Silva Simao 2006], the authors propose a similarity-based approach to select a subset of tests from a test set in the context of regression testing. They apply a model to classify test cases by using an ART-2A (Adaptive Resonance Algorithm) self-organizing neural network architecture. Here, we will not give a description of the algorithm, interesting readers could refer to

[Carpenter 1991] for details. We will just explain the principle of the authors approach. The model applied uses structural coverage information of test cases such as their source code statement coverage. The classification consists in grouping similar test cases in the same cluster. The same cluster contains tests that explore the same software test characteristics. When the software is modified, the authors apply an automatic tool to identify the points in source code where changes have been applied and selects the most adequate subset of test from clusters to represent the software modification.

In [Masri 2007], the authors propose a similarity-based approach to select test cases based on their execution profiles. A profile is defined as test execution characterization that indicates the frequency of execution of certain program elements, that are considered relevant to make an execution succeed or fail. The diversity technique proposed observes how these execution profiles are distributed using a dissimilarity function, that takes a pair of profiles and gives a real number representing their degree of dissimilarity. The metric used by authors consists in comparing two profiles based on the number of times profile features were exercised. Afterwards, a clustering technique is used to group tests into clusters based on the dissimilarity measure. A set of tests are selected from each cluster or from a particular cluster by applying a sampling method. For example, one-per-cluster sampling method selects randomly one test from each cluster.

In this section, we presented two similarity test reduction approaches based on code-level information. In next section we present some similarity test reduction approaches based on model-level information.

### 9.3.2 Model-based similarity functions for test reduction

In [Cartaxo 2011], the authors approach is proposed for Labelled Transition System (LTSs) models from which test cases are generated. To apply the approach, the LTS behavior model and the intended percentage of path coverage have to be specified by the tester. A test case is considered as a path in the LTS model. To measure the similarity between two paths in an LTS, the number of identical transitions between them is counted. Identical transitions are those having exactly the same source and target states and the same label. The similarity function is denoted as Identical Transitions Similarity (It). The similarity function between two test cases $tc_i$ and $tc_j$ is equal to the number of identical transitions in $tc_i$ and $tc_j$ divided by average between $tc_i$ and $tc_j$ paths length. One test is removed from the pair with highest degree of similarity. The removed test should have the smallest number of transitions. If the number of transitions is equal between the two tests a random choice is then done. This process is repeated for each pair of tests until the intended

path coverage is reached.

In [Hemmati 2010], the authors propose three different encodings for a test path in UML state machine: state-based, transition-based and triggered-guard-based. Similarity value can be then calculated based on the encoding. Two types of similarities are considered: set-based similarity an sequence-based similarity. The set based similarity is defined on two sets of elements and sequence-based similarity is defined on two sequences of elements. The difference between the two similarities is whether the order is taken or into account or not. For example, if a test case $tc_1$ includes the sequence of operation calls opA; opB and a test case $tc_2$ contains the sequence opB; opA, these two test cases are considered as equivalent according to the set-based similarity. Given a similarity function (SimFunc) and a set of encoded test cases (Sn) with $S_n \neq \emptyset$, the test selection problem consists in minimizing SimMsr (Sn).

$$SimMsr(S_n) = \sum_{tp_i, tp_j \in Sn \wedge i > j} SimFunc(tp_i, tp_j)$$

Other authors propose similarity functions not to reduce test cases but to prioritize them. Test prioritization does not remove test cases from the tests set like Test reduction. It allows only to order their execution so that tests which are more important (according to some measure) are executed first. We thought interesting to present these approaches because they also can be applied for test reduction.

### 9.3.3 Similarity functions for test prioritization

Ledru et al. [Ledru 2009] propose an approach to prioritize test cases using a similarity function based on the test cases scripts. The approach can be applied for the model based and code based context since it is based on test strings.

The similarity function calculates String distances between test cases as the Hamming distance or the Levenshtein distance. The distance is computed between a test case $t_c$ and a set of test cases TS. It is the minimum distance over the distances calculated between the test case $t_c$ and each test case from TS.

The test cases prioritization algorithm is performed using a greedy algorithm to choose at each iteration the test case which is the most distant from the test set already selected. The advantage of this approach is that it does not require the model or the code of the system under test, but it uses only the textual information of the test cases. However string based similarity does not take into account semantics difference between test cases, that may have an influence on test case execution.

In [Jiang 2009], authors propose an approach for test case prioritization by using a similarity function based on coverage information. The next test case to select is the most diverse one comparing to the test cases already prioritized. The test cases are first associated with the set of statements covered in their execution. The distance between the two test cases is measured using the Jaccard distance based on the two sets. The distance between two sets A and B is equal to $D(A, B) = 1 - \mid A \cap B \mid / \mid A \cup B \mid$. If the two sets are equal the distance will be equal to zero.

To perform test case prioritization, Yoo et al. [Yoo 2009] propose an approach based on a clustering technique. The clustering strategy puts a set of objects into different groups, where each group contains objects with common properties. To define which properties are used to measure the similarity of objects, a clustering criterion is used. The best criterion that can be used is the faults detected by the tests. However, the faults detected can be known only after test execution possibly on a population of mutants (a mutant is a program where a set faults are inserted). Therefore, the criterion to choose should be a relevant substitute that clusters similar test cases to likely detect the same faults. In [Yoo 2009], the authors use dynamic execution traces as a clustering criterion. Each test case execution generates a trace presented as a string of binaries. The bit corresponds to a statement in the program. The binary is equal to 1 if the statement is executed, and 0 otherwise. The Hamming distance is then applied between two binary strings to measure the similarity between two tests. Using the similarity measure, two types of prioritization are realized, the intra-cluster prioritization and the inter-cluster prioritization. Intra-cluster prioritization consists in prioritizing test cases inside a cluster. Inter-cluster prioritization consists in prioritizing clusters where each cluster is represented by a test case (the first one according to Intra-cluster prioritization) . The selection of test cases begins from the cluster with highest priority. The selection of test is switched to another cluster, when the next selected test inside the same cluster does not improve the number of faults detected.

## 9.4 Conclusion

In this chapter we presented two big families of approaches. The first one uses different algorithms to reduce the test set based on coverage of code (or requirements) in a way to maximize the coverage of code among tests. The second family of approaches defines a similarity function based on information related to the test (as test execution, test string). This function is used to calculate a diversity measure among tests and then basing on this measure

keeps in the test set the tests the most diverse. We have seen some similarity functions applied for test prioritization, to order test execution. The test reduction may have a loss in fault detection capability as shown in [Jones 2001]. The use of one technique instead of another one to perform test reduction depends of many factors. For example, if we are in a model-based context we use techniques different from the ones used in code-based context. It depends also of the availability of the algorithms implementation and the spatial/temporal complexity of executing them on the resources used.

In our work, we proposed a new test reduction technique that takes into account some information that other test reduction techniques do not. The test reduction techniques based on code (specification or requirements) coverage studied in the literature do not consider the order/repetition of execution of some elements in the code. We found only the work of Masri *et al.* that uses a criterion for reducing a test set based on the number of times an element of the code is covered by execution. Another popular test criterion reported in literature that takes into account these two information: order and repetition of code instructions, is the path coverage criterion. However, the path coverage criterion is a strong criterion to decide on similarity between two tests. We thought that many similarity relations can be defined basing on order and repetition of covering some elements in the code. Therefore, we proceed by annotating the code by inserting tags at the points we consider important. The test is then executed and the tags covered by the test are saved. Basing on the order/repetition of tags covered we define different equivalence relations used to decide on the equivalence of two test cases. The choice of an equivalence relation instead of another depends of the testing needs.

In the next chapter, we present our approach proposed to reduce a test suite using similarity relations based on covered information.

Chapter 10

# Test suite reduction using equivalence relations

## Contents



## 10.1 Introduction

In this chapter, we propose an approach to reduce a test suite based on several equivalence relations. The test reduction approach exploits the availability of existing information called *tag* inserted in the code/specification for various purposes. After executing/animating a test suite, a trace is collected containing the tags covered. A test case represents a sequence of operation calls. Executing an operation call generates a sequence of tags. Executing a test case gives then a sequence of tags sequences. The equivalence relations differ in the way they consider or not the order/repetition of tags. The weakest relation considers two test cases as similar if they cover the same set of tags. The strongest one considers them as equivalent if they cover exactly the same sequence of tags sequences. Four equivalence relations are defined.

In Sect. 10.3, we propose a family of four equivalence relations based on annotation traces and covering a range of permissive to strict equivalences.

Sect. 10.4 describes how the reduction is performed on a test suite using the proposed equivalence relations and illustrates our test reduction approach on an example of a test suite. Sect. 10.5 presents an extension proposed for the reduction algorithm.

Finally, Sect. 10.7 draws the conclusions and perspectives of this work.

## 10.2 Code Annotation

### 10.2.1 Principle

Our approach relies on the availability of annotations inserted at different points in the source code or in the specification of the SUT. We called these annotations as tags and they are intended to be covered by executing the program to collect a trace. The insertion of tags may be motivated by various concerns:

- Source code instrumentation [Geimer 2009, Zhang 2011]: it consists of inserting instructions in the source code to trace system execution. The resulting trace may serve various purposes such as debugging, profiling or code coverage measurement.

- Traceability: these annotations can be used to trace back system/user requirements. This is useful to ensure that all requirements in the requirement specification document are implemented in the source code and tested by at least one test case [Connolly 2009, Mei 2009]. In our research work, we used OCL specifications to animate test cases with CertifyIt tool. In these specifications several tags are inserted to trace back system requirements. We will see in Chapter 11 (Experimentations) an evaluation of our approach on an OCL specification using tags.

In the sequel we illustrate our approach on tags inserted in source code. The illustrations using tags inserted in specification will be presented in the next chapter. To better understand our approach we first present some technical details about our tagging system.

We use a code annotation system based on calls to a static class named *TagLogger*. Fig. 10.1 presents a simple example of an annotated class. A class *Value* stores an integer, named *val*, which can be incremented or decremented by operations *inc* and *dec*. These operations take as argument an array of integers, and add or substract to *val* the absolute value of each element of the array. The code of *inc* and *dec* includes calls to the *Taglogger* class. For space reason, we only show method *inc* in Fig. 10.1.

```
public class Value {
 int val=0;
 public void inc(int[] intTab) {
  TagLogger.beginOpCall();
  for(int i=0; i<intTab.length; i++){
   int x = intTab[i];
   if (x > 0) {
    val = val + x;
    TagLogger.log("Inc-gt0");
   }
   else if (x < 0) {
    val = val + Math.abs(x);
    TagLogger.log("Inc-lt0");
   }
   else {
    TagLogger.log("eq0");
   }
  }
  TagLogger.endOpCall();
 }
...  // Code for dec method
}
```

| Log file |
|---|
| 1_1_1:Inc-gt0 |
| 1_1_2:Inc-lt0 |
| 1_2_1:Dec-lt0 |
| 1_2_2:Dec-gt0 |
| 2_1_1:Inc-gt0 |
| 2_1_2:Inc-lt0 |
| 2_2_1:Dec-lt0 |
| 2_2_2:Dec-gt0 |

```
G(tc1) =
 [[Inc-gt0, Inc-lt0],
  [Dec-lt0, Dec-gt0]]

G(tc2) =
 [[Inc-gt0, Inc-lt0],
  [Dec-lt0, Dec-gt0]]
```

Figure 10.1: Annotated Java class and example of tagging log file for the test cases of Fig. 10.2

Taglogger records the tags covered by the execution of the class. For instance the instruction *TagLogger.log("Inc-lt0")* denotes that a tag *"Inc-lt0"* is activated and recorded in a file. We used two instructions *TagLogger.beginOpCall()* and *TagLogger.endOpCall()* to indicate the begin and the end of the tagged method. We used *TagLogger.beginTestCase()* to indicate the transition from a test case execution to another. The logging system records tags with three elements: the test case number, the operation call number where the tag is activated, and the tag number. This information is useful to know tag activation order in execution.

Consider the two test cases in Fig. 10.2. The execution of the ValueTest class produces a logging file (Fig. 10.1) that contains the tags activated. A tag is recorded using the following pattern: *tcNum_opNum_tagNum:tagName*. For example, *1_1_2:Inc-lt0* tells us that *Inc-lt0* was the second tag activated in the first operation call of the first test case (tc1). In Fig. 10.1, $\mathcal{G}(\texttt{tc}_i)$ with i=1 or 2 represents the sequence of tags sequences resulting from the execution of the test case $\texttt{tc}_i$.

```
import logging.TagLogger;
import org.junit.Test;

public class ValueTest extends TestCase {
 public void tc1(){
  TagLogger.beginTestCase();
  Value val = new Value();
  val.inc(new int[]{1,-6});
  val.dec(new int[]{-2,4});
 }
 public void tc2(){
  TagLogger.beginTestCase();
  Value val = new Value();
  val.inc(new int[]{4,-2});
  val.dec(new int[]{-1,2});
 }
}
```

Figure 10.2: Example of JUnit test cases executing an annoted Java class

We can observe in the logging file of Fig. 10.1 that the test cases tc1 and tc2 produce the same sequence of tags. Therefore, they are equivalent according to an equivalence relation that compares the tag sequences.

In the case of nested operation calls, i.e. a tagged operation(with beginOpCall) that calls another tagged operation (with beginOpCall), the logging system records the tags covered in the nested operation calls but without creating a new sequence of tags. The system records the tags as if they were covered for the root operation (that exists in the test case). For instance consider a test case tc3 with `tc3 = op1();`. Suppose that the execution of `op1` covers two tags t1 and t2, and then calls op2 that covers t3 and t4. The result generated will be:
$\mathcal{G}$`(tc3) = [[t1, t2, t3, t4]]`
We see that all the tags (t1, t2, t3 and t4) are recorded as if they were covered in a single operation.

We assume that the tags activation system has no side effects and is correctly implemented.

### 10.2.2 Tagging process

The insertion of tags can proceed manually or automatically. In the manual tagging process, the user inserts tags at the points he considers relevant to trace. At these points, execution of the program is likely to affect the result or to change the system state, for example, one may tag a conditional branch where an attribute value is modified. The points that only display an information are not impacting the output and then are not candidates for tags insertions. The places where tags are inserted can be after a set of instructions, inside conditional branches, inside loops, inside exceptional processes, after a method call. A tag is related to a set of instructions. For example consider a permutation process of two variables (x and y) using a third variable (z). A tag can be defined for the permutation process and is inserted after the execution of the process instructions. The process is done in three instructions after which the tag is inserted: z:=x; x:=y; y:=z; @tag_permutation In our approach, the tag is always inserted after the related instructions are executed to ensure that all of them have a successful execution. In our case, the test suite is executed before launching the reduction algorithm. Then, it is important to make a difference between a failing test and a non failing test by the way we insert our tags. For example if a unhandled exception (i.e. in Java code) is triggered, the tag must not be activated because some of the related instructions are not executed. For a handled exception, what makes the difference between a normal execution of the tags related instructions and its exceptional execution is the activation of the tag defined for the exception.

In the manual tagging process, the user aims to maximize the diversity among the test suite by creating a fine-grained tagged system. The more he creates tags in the code, the more the trace will have precise behavior activation information and the more it is possible to assess the diversity of test cases.

Automatic processes for tags insertions are used in the literature for automatic requirements traceability between the requirement document, the model and the source code [Antoniol 2002, Cleland-Huang 2007]. The automatic insertion of tags is limited to the tags related to the element defined in the model such as the classes, the fields and the methods. In these works, the automatic insertion of tags to trace a fine grained requirement is not available. A fine grained requirement can be represented as a set of instructions inside a method, or a specific behavior of the method.

In the next section, we present the equivalence relation family proposed to reduce a test suite.

## 10.3 Equivalence relations

Consider a test suite $TS$, defined as a set of test cases and a test case $TC$ as a sequence of method calls. We denote $\mathcal{G}(op)$ the *sequence* of tags obtained by a given execution of a method $op$. We denote a set of elements between "{" and "}", and a sequence of elements between "[" and "]". The *set* of tags associated to a method call is denoted $\mathcal{S}(\mathcal{G}(op)) = \{tg_i \mid tg_i \in \mathcal{G}(op)\}$. The execution of a test case $TC = [op_1, op_2, ..., op_m]$ results in a sequence of tag sequences, which is denoted $\mathcal{G}(TC) = [\mathcal{G}(op_1), \mathcal{G}(op_2), ..., \mathcal{G}(op_m)]$. $g_{Ta} = \bigcup_{op_i \in T_a} \mathcal{S}(\mathcal{G}(op_i))$ denotes the union of all tags set ($\mathcal{S}(\mathcal{G}(op_i))$) generated from operation calls ($op_i$) of test case ($T_a$).

Based on the sequences of tag sequences obtained by the execution of the tests, it is possible to define a family of equivalence relations $R_i$. For a given relation $R_i$, we write $TC_j \equiv_{R_i} TC_k$ to assert that test cases $TC_j$ and $TC_k$ are equivalent according to relation $R_i$.
In what follows, we use the example described in the previous section to illustrate each equivalence relation.

The weakest equivalence relation of our family is $R_0$.

**Definition 1 (Equivalence of tag set ($R_0$))** *Two test cases $T_a$ and $T_b$ are equivalent according to a tag set (noted $T_a \equiv_{R_0} T_b$) iff the same tag set is collected during both test executions.*
*Let* $g_{Ta} = \bigcup_{op_i \in T_a} \mathcal{S}(\mathcal{G}(op_i))$ *and* $g_{Tb} = \bigcup_{op_j \in T_b} \mathcal{S}(\mathcal{G}(op_j))$
*Then* $T_a \equiv_{R_0} T_b \Leftrightarrow g_{T_a} = g_{T_b}$

The equivalence relation $R_0$ establishes an equivalence between tests based on observed tag sets. It does not take into account the possible repetition or ordering of tags, since it relies on the notion of set. This relation is reflexive ($T \equiv_{R_0} T$), symmetrical ( $T \equiv_{R_i} T' \iff T' \equiv_{R_0} T$) and transitive ($T \equiv_{R_0} T' \wedge T' \equiv_{R_0} T'' \implies T \equiv_{R_0} T''$). It is thus an equivalence relation.

Consider a test suite TS with three test cases $TC_1$, $TC_2$ and $TC_3$:
$TC_1$ = val.inc([1,-6,2]); val.dec([0]);
$TC_2$ = val.inc([-3,2,0]); val.inc([]);
$TC_3$ = val.inc([1,-6]); val.dec([3,1]);
$\mathcal{G}(TC_1)$ = [[Inc-gt0,Inc-lt0,Inc-gt0], [eq0]]
$\mathcal{G}(TC_2)$ = [[Inc-lt0,Inc-gt0,eq0], []]
$\mathcal{G}(TC_3)$ = [[Inc-gt0,Inc-lt0], [Dec-gt0,Dec-gt0]]
$g_{TC_1}$ = {Inc-lt0,Inc-gt0,eq0}
$g_{TC_2}$ = {Inc-lt0,Inc-gt0,eq0}

$g_{TC_3}$ = {Inc-gt0,Inc-lt0,Dec-gt0}

$TC_1$ and $TC_2$ are equivalent according to $R_0$ because $g_{TC_1} = g_{TC_2}$. However, $TC_3$ is not equivalent to $TC_1$ and $TC_2$ because $g_{TC_3}$ differs from $g_{TC_1}$ and $g_{TC_2}$.

A second equivalence relation keeps the individual traces of each operation execution as the set of activated tags. This relation considers the set of tag sets produced by each test case without mattering of their execution order.

**Definition 2 (Equivalence of set of tag sets ($R_1$))** *Two tests $T_a$ and $T_b$ are equivalent according to a set of tag sets (noted $T_a \equiv_{R_1} T_b$) iff the same set of tag sets is collected during both test executions.*
*Let $s_{Ta} = \{\mathcal{S}(\mathcal{G}(o_i)) \mid o_i \in T_a\}$ and $s_{T_b} = \{\mathcal{S}(\mathcal{G}(o_j)) \mid o_j \in T_b\}$ Then*
$T_a \equiv_{R_1} T_b \Leftrightarrow s_{T_a} = s_{T_b}$

Let us now consider $TC_4$ and $TC_5$ as follows:
$TC_4$ = val.inc([1,2]); val.inc([0]); val.dec([0]); val.inc([-2,1]);
$TC_5$ = val.inc([0]); val.inc([3,-3]); val.inc([3]);
$\mathcal{G}(TC_4)$ = [[Inc-gt0,Inc-gt0], [eq0], [eq0], [Inc-lt0,Inc-gt0]]
$\mathcal{G}(TC_5)$ = [[eq0], [Inc-gt0,Inc-lt0], [Inc-gt0]]
$s_{TC1}$ = {{Inc-gt0,Inc-lt0}, {eq0}}
$s_{TC4}$ = {{Inc-gt0},{eq0}, {Inc-gt0,Inc-lt0}}
$s_{TC5}$ = {{eq0}, {Inc-gt0,Inc-lt0}, {Inc-gt0}}
$s_{TC2}$ = {{Inc-lt0,Inc-gt0,eq0}, {}}
$g_{TC4} = g_{TC5}$ {Inc-lt0,Inc-gt0,eq0}

We observe that tests $TC_4$ and $TC_5$ produce the same sets of tag sets. Therefore, $TC_4 \equiv_{R_1} TC_5$. $TC_1$ and $TC_2$ do not give the same sets of tags sets as for $TC_4$ and $TC_5$, however these four tests are equivalent according to $R_0$ because $g_{TC1} = g_{TC2} = g_{TC4} = g_{TC5}$.

$TC_4$ and $TC_5$ do not produce exactly the same sequences of tag sets. Since the execution order of the methods in the test case may have an influence on the tested behavior, we introduce a third equivalence relation, which discriminates tests on the basis of sequences of tag sets.

**Definition 3 (Equivalence of sequences of tag sets ($R_2$))** *Two tests $T_a$ and $T_b$ are equivalent according to a sequence of tag sets (noted $T_a \equiv_{R_2} T_b$) iff the same sequence of tag sets is collected during both test executions.*
*Let $ss_{Ta} = [\mathcal{S}(\mathcal{G}(o_i)) \mid \forall\, o_i \in T_a]$ and $ss_{Tb} = [\mathcal{S}(\mathcal{G}(o_j)) \mid \forall\, o_j \in T_b]$ Then*
$T_a \equiv_{R_2} T_b \Leftrightarrow ss_{Ta} = ss_{Tb}$

Let consider $TC_6$ as follows:
$TC_6$ = val.inc([0]); val.inc([-6,2]); val.inc([3,4]);
$\mathcal{G}(TC_6)$ = [[eq0], [Inc-lt0,Inc-gt0], [Inc-gt0,Inc-gt0]]
$s_{TC6}$ = {{eq0}, {Inc-gt0,Inc-lt0}, {Inc-gt0}}
$ss_{TC4}$ = [{Inc-gt0},{eq0}, {eq0}, {Inc-gt0,Inc-lt0}]
$ss_{TC5}$ = [{eq0}, {Inc-gt0,Inc-lt0}, {Inc-gt0}]
$ss_{TC6}$ = [{eq0}, {Inc-lt0,Inc-gt0}, {Inc-gt0}]

As shown previously, $TC_4$ and $TC_5$ are equivalent according to $R_1$. This is not true according to $R_2$. $TC_6$ and $TC_5$ produce the same sequence of tag sets, that is why both tests are equivalent w.r.t. $R_2$.

However, $TC_6$ and $TC_5$ do not have the same sequences of tag sequences ($\mathcal{G}(TC_5) \neq \mathcal{G}(TC_6)$). To differentiate the two tests in this case, we introduce $R_3$, that requires the same sequences of tag sequences.

**Definition 4 (Equivalence of sequence of tag sequences ($R_3$))** *Two tests $T_a$ and $T_b$ are equivalent according to a sequence of tag sequences (noted $T_a \equiv_{R_2} T_b$) iff the same sequence of tag sequences is collected during both test executions.*
$T_a \equiv_{R_0} T_b \Leftrightarrow \mathcal{G}(T_a) = \mathcal{G}(T_b)$

We remind that:
$\mathcal{G}(TC_6)$ = [[eq0], [Inc-lt0,Inc-gt0], [Inc-gt0,Inc-gt0]]
and $\mathcal{G}(TC_5)$ = [[eq0], [Inc-gt0,Inc-lt0], [Inc-gt0]]

Let $TC_7$ = val.inc([0]); val.inc([-3,5]); val.inc([9,6]);
with $\mathcal{G}(TC_7)$ = [[eq0], [Inc-lt0,Inc-gt0], [Inc-gt0,Inc-gt0]]
we have $TC_6 \equiv_{R_3} TC_7$.

We have formalized four equivalence relations. The discriminating power of the relations increases from $R_0$ to $R_3$, i.e. $R_0 \leq R_1 \leq R_2 \leq R_3$.

A first advantage of our approach is that it is possible to adjust the size of the reduced suite by choosing a stronger or weaker equivalence relation. We illustrate the influence of the equivalence relation on the size of test suite in the next chapter. Another advantage is that the approach does not rely on a specific annotation scheme. Annotations may result from source code instrumentation process or from traceability purposes. Of course, the accuracy/completeness of annotations will impact the relevance of the reduced test suite.

As any test reduction approach based on an analysis of coverage, it requires to run at least once every test case. This will not be perceived as an inconvenience in many projects where the test suite is executed regularly.

## 10.4 Reduction process

To generate a reduced test set, we first run the original test set. A logging file is generated storing for each test case $TC_i$, a sequence of activated tag sequences $\mathcal{G}(TC_i)$. Then, we execute the algorithm of reduction to get the reduced test set. The algorithm takes as input all the generated sequences of tags sequences (where each sequence of tags sequence corresponds to a test case $TC_i$ in the test suite) and the equivalence relation.

The reduction algorithm is performed as follows. In the first iteration, the first test case is compared to all the other test cases. The equivalent test cases are grouped in the same equivalence class. In the second iteration, the next non classified test case is compared to the remaining non classified test cases, and puts the equivalent test cases in another equivalence class. If a test case is not equivalent to any other test case, it is put alone in an equivalence class. The iteration proceeds until classifying all equivalent test cases. Finally, the algorithm selects randomly a test case from each equivalence class to get the reduced test set.

To illustrate the algorithm, let us take the test set TS that contains the 7 test cases defined in Sect. 10.3 ($TC_1$ to $TC_7$). We present in Fig. 10.3 for each test case $TC_i$ the corresponding sequence of tags sequences $\mathcal{G}(TC_i)$.

The reduction algorithm using the equivalence relation $R_0$ is performed as follows. First, $TC_1$ is compared to all other test cases. An equivalent class is created grouping all the equivalent test cases ($TC_1$, $TC_2$, $TC_4$, $TC_5$, $TC_6$ and $TC_7$). Next, the remaining test $TC_3$ is put into a new equivalence class. Finally, the algorithm selects randomly a test case from each equivalence class. For example, the reduced test set can be $\{TC_1, TC_3\}$.

The algorithm of reduction is applied on TS using the four equivalence relations, and we get the following reduced test sets $RTS_{R_i}$ (reduced test set according to the equivalence relation $R_i$):
$RTS_{R_0}$ = {TC1, TC3} : size = 2
$RTS_{R_1}$ = {TC1, TC2, TC3, TC4, TC5} : size = 5
$RTS_{R_2}$ = {TC1, TC2, TC3, TC4, TC5} : size = 5
$RTS_{R_3}$ = {TC1, TC2, TC3, TC4, TC5, TC6} : size = 6
We can see that $RTS_{R_0} \subseteq RTS_{R_1} \subseteq RTS_{R_2} \subseteq RTS_{R_3}$ due to the discriminating power that increases from R0 to R3.

In next section, we present an extension proposed for the algorithm.

| i | $TC_i$ | $\mathcal{G}(TC_i)$ |
|---|---|---|
| 1 | inc([1,-6,2]); dec([0]); | [[Inc-gt0,Inc-lt0,Inc-gt0], [eq0]] |
| 2 | inc([-3,2,0]); inc([]); | [[Inc-lt0,Inc-gt0,eq0], []] |
| 3 | inc([1,-6]); dec([3,1]) | [[Inc-gt0,Inc-lt0], [Dec-gt0,Dec-gt0]] |
| 4 | inc([1,2]); inc([0]); dec([0]); val.inc([-2,1]); | [[Inc-gt0,Inc-gt0], [eq0], [eq0], [Inc-lt0,Inc-gt0]] |
| 5 | inc([0]); inc([3,-3]); inc([3]); | [[eq0], [Inc-gt0,Inc-lt0], [Inc-gt0]] |
| 6 | inc([0]); inc([-6,2]); inc([3,4]) | [[eq0], [Inc-lt0,Inc-gt0], [Inc-gt0,Inc-gt0]] |
| 7 | inc([0]); inc([-3,5]); inc([9,6]); | [[eq0], [Inc-lt0,Inc-gt0], [Inc-gt0,Inc-gt0]] |

Figure 10.3: Example of test suite TS to reduce and its execution trace

## 10.5 Extension to the reduction algorithm

We propose an extension to the reduction algorithm aiming to improve the reduction rate for R2 and R3. The reduction algorithm considers two test cases equivalent according to R2 and R3 only if the two test cases have the same length (number of operation calls). Using this equivalence principle, many test cases existing in the reduced test suite may be prefixes for other test cases. Our idea is then to delete these test cases (prefixes) since their executions are included in other test cases. In our proposal, we consider a test case (denoted as "small" test case) as a prefix of another test case if its sequence of tags sequence generated is a prefix in a sequence of tags sequence of another test case (denoted "large" test case). We give an example of 2 test cases TC1 and TC2 with: $\mathcal{G}(TC_1)$ = [[t1,t2], [t3,t4]] and
$\mathcal{G}(TC_2)$ = [[t1,t2], [t3,t4], [t5]].
In this example TC1 is considered as prefix of TC2 because the sequence of tags sequences [[t1,t2],[t3,t4]] is included in $\mathcal{G}(TC_2)$. The extension of reduction algorithm is applied for the equivalence relations R2 and R3 where the activation order of tags sets/sequences of an operation call makes sense in the operation call sequence. It consists in finding all the prefixes (sequences of tags sets/sequences) included in other sequences of tags sets/sequences and delete the corresponding test cases. We call R2' or R3', the reduction process that consists in using the equivalence relation R2 respectively R3 to reduce a test suite and then applying the prefix deletion process. The motivation behind the deletion of prefixes is that we consider that the system behavior covered by small test cases execution is covered by larger test cases execution, and therefore, executing the large test case is sufficient.

In the next chapter we present experimental results of our test suite reduction approach performed on different case studies. We use the standard algorithm (without extension) for the evaluation of examples. An example of evaluation using the algorithm with extension is presented in Sect. 11.2.3.

## 10.6 Comparison with traditional approaches for structural based reduction

The readers can say that our approach is similar to the structural reduction approaches because they depend both on the structure of code. For example, to compare equivalence of test cases based on their branch coverage, a tag is inserted at each code branch. For instructions coverage we insert a tag after each instruction bloc. Then, the recorded trace is compared between tests, and the reduction is performed when two tests cover the same elements

related to the chosen coverage criteria (i.e. branches, instructions, execution path, etc.).

Many algorithms (as the HGS algorithm) have been proposed to perform the test suite reduction based on the coverage information. It takes a test suite and an array containing for each test the set of elements it covers, and provides a reduced test suite representative of the original one.

It is true that our approach is similar to structural reduction approaches when the tags are inserted in the code to perform source code instrumentation and to compare it to other approaches based on structure. However in the case where the tags are inserted in the code for requirements traceability, the annotation process is different. Not every branch/instruction is tagged, only the points that trace back a specific user requirement are tagged. Moreover, the tags inserted in the source code to trace user requirements may be not unique. They can be used more than one time at different points. This can be explained by the fact that many points in the source code are realizing the user requirement. For example a process of specific exceptional behavior can be the same in many functions and therefore tagged with the same name of tag. Another example, adding a node to a tree at the left or adding it at the right can be tagged by a single tag: `add_tag`.

Another difference between our approach and structural reduction approaches is the relation of equivalence used to find diversity among tests. The structural approaches are using traditional testing coverage criteria as branch coverage, instruction coverage or method coverage. These criteria (except for path coverage criterion) do not consider the number of repetitions of an instruction or an operation call. They do not consider the execution order of an instruction or an operation call. They see only if instructions/branches/methods are covered or not. In our approach, 4 equivalences relations are proposed based on two important criteria: the order and the repetition of tags. The order and repetition of tags can be considered inside an operation call or between two operation calls.

It is important to take into account the order and repetition of tags inside a method when the execution order of instructions inside a method makes sense. In this case, the execution of the instructions related to the $tag_i$ can affect the execution of subsequent instructions related to the $\text{tag}_{i+1}$. Let us consider the example of code of operation `opOnArray` in Fig. 10.4. The operation iterates on the integer elements of an array. If the element (T[i]) is even then we divide the variable S by T[i] (the tag DIV2 is activated), otherwise we multiply it by T[i] (the tag MUL2 is activated). The operation return then s. In this operation the order of covering the branches in the iteration (activation of tags) is important and gives a different result each time we change the order. If we define two test cases $\text{tc}_A$ and $\text{tc}_B$ with:

```
public int opOnArray (int[] T){
  TagLogger.beginOpCall();
  int s=10;
  for(int i=0; i<T.length; i++){
    if(T[i]%2==0){
      s=s / T[i];
      TagLogger.log("DIV2");
    }
    else{
      s=s * T[i];
      TagLogger.log("MUL2");
    }
  }
  TagLogger.endOpCall();
  return s;
}
```

Figure 10.4: Example to motivate the importance of order and repetition of tags

$tc_A$: opOnArray([4,3]) –> s=6
$tc_B$: opOnArray([3,4]) –> s=7
$\mathcal{G}$(tcA) = [[DIV2, MUL2]]
$\mathcal{G}$(tcB) = [[MUL2, DIV2]]
If we consider the order of activation of tags, the two test cases $tc_A$ and $tc_B$ will be not equivalent. This is motivated by the fact that they give different results for s by changing the order of tags activations. The two tagged points (branches) are dependent because they modify the same variable.

The order is not important when our target is only to activate the tags independently of their order of activation. The loop instructions, the recursion and the goto statements are the types of instructions that give different orders and numbers of repetitions of tags.

## 10.7 Conclusion

The concept of source code annotations is used for a variety of purposes, from traceability [Connolly 2009, Mei 2009] to code instrumentation [Geimer 2009, Zhang 2011]. We propose to take advantage of these annotations for test suite reduction. We define four equivalence relations between tests based on the annotations covered during test execution. These relations define equivalence

classes for tests which are used to extract the reduced test suite. Depending on needs and/or time limit, it is possible to adjust the size of reduced test suite by choosing the most appropriate equivalence relation.

Chapter 11

# Experimentation

## Contents



## 11.1 Introduction

This chapter gives the results of experimenting our test reduction approach for different test suites and on different case studies. Sect. 11.2 presents the test suite reduction result using the four equivalence relations on small programs. The tags are inserted in the programs for code instrumentation purpose. The reduction rate and the fault detection capability are compared between the reduced test suites and to the random selection approach. To compute the faults detected by a test case we use the mutation testing by inserting mutations into the programs under test.

Sect. 11.3 presents the experimental results of our approach on a case study (Video On Demand Player) where tags are inserted for requirements traceability purpose. The reduction rate and the program coverage percentage are compared between the reduced test suites using the four equivalence relations and to the original test suites.

Sect. 11.4 presents how our approach can be used in the context of annotated specifications. An experimentation is performed on a specification of an electronic purse application containing requirement traceability annotations. The collection of tags covered is performed during test animation using CertifyIt. Using these collected tags our tool is then able to reduce the test suite after its animation.

## 11.2 Preliminary Experimentations

| Program Under Test | LOC | #Methods | #tag | #Mutants |
|---|---|---|---|---|
| ArrayStack [Weiss 2007] | 100 | 8 | 17 | 54 |
| AvlTree [Weiss 2007] | 281 | 18 | 43 | 114 |
| BinarySearchTree [Weiss 2007] | 219 | 15 | 42 | 166 |
| BinomialHeap [Weiss 2007] | 434 | 6 | 20 | 73 |
| BinomialQueue [Weiss 2007] | 222 | 14 | 21 | 94 |
| BoundedStack [Li 2009] | 75 | 10 | 15 | 167 |
| Buffer [Ledru 2004] | 44 | 4 | 9 | 156 |
| Node [Li 2009] | 136 | 9 | 15 | 15 |
| Queue [Li 2009] | 73 | 5 | 10 | 71 |
| RedBlackTree [Weiss 2007] | 254 | 16 | 29 | 71 |
| VendingMachine [Li 2009] | 85 | 6 | 13 | 104 |

LOC is computed with *LOC Calculator* tool, `http://code.google.com/p/loc-calculator/`.

Figure 11.1: Subjects of the experimentation

We led several small experiments to investigate the influence of the equivalence relations on the size and effectiveness of the reduced test suite. We expect that a weaker equivalence criterion will lead to smaller test suites than the stronger ones. We also compare the effectiveness of the reduced against the original test suites.

In this section, we measure the effectiveness of a test suite by evaluating its fault detection capability. To evaluate the fault detection capability of the test suites, we introduce faults in the program under test following a classical mutation approach [Offutt 1994]. Faults are introduced in copies of

the program under test; each erroneous copy contains only one fault and is called *mutant*. Faults are inserted with respect to a *fault model*, expected to be representative of real faults. If a difference can be observed between the execution of the original program and the execution of the mutated one, the mutant is *killed*. It is *alive* otherwise. In the sequel, we will compare the number of mutants killed respectively by the original and the reduced test suites.

### 11.2.1 Experimental setting

*Subjects* We collected already existing subjects that are involved in other experimental studies. For instance, we selected some container classes used in [Weiss 2007, Li 2009] and other types of classes [Li 2009, Ledru 2004]. Eleven classes were thus analyzed. The table given in Fig. 11.1 provides some structural characteristics of these classes. Our experiment concerns 33 test suites, 3 test suites for each program. The test suites were generated independently of this study. They were generated using a combinatorial tool developed by Lydie du-Bousquet a member of our research team.

*Annotation process* Tags were inserted in the source code manually, but systematically. They record the execution flow and are located:

- after instruction blocks that modify the state of the system,
- inside iteration blocks,
- inside conditional branches,
- inside class constructors,
- after operation calls (including recursive calls),
- inside exception block processing.

*Faults* were introduced in the classes under test using the MuClipse tool[1][Smith 2009]. MuClipse proposes two types of mutation operators: traditional or class-level mutant operators. Class-level mutants are dedicated to evaluate tests with respect to classical mistakes in the use of object-oriented features, such as inheritance or polymorphism. We choose to consider only traditional mutants, since the programs under tests are not implemented using inheritance or polymorphism. Traditional mutations include replacing operands, deleting statements, replacing some arithmetic operation with another one (like "+" by "-"), replacing some variable with another one, ... To

[1] http://muclipse.sourceforge.net/

| Original TS | | Test suite size | | | | | The average numbers of killed mutants | | | | | | | | | Mutants |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | with $R_0$ | | with $R_1$ | | with $R_2$ | | with $R_3$ | | missed by |
| SUT | TS0 | $\mid TS0 \mid$ | $\mid TS_{R_0} \mid$ | $\mid TS_{R_1} \mid$ | $\mid TS_{R_2} \mid$ | $\mid TS_{R_3} \mid$ | TS0 | TB-TS | R-TS | TB-TS | R-TS | TB-TS | R-TS | TB-TS | R-TS | TB-TS($R_3$) |
| Vending Machine | 1 | 7 | 5 | 5 | 5 | 5 | 1.0 | 1.0 | 0.93 | 1.0 | 0.91 | 1.0 | 0.89 | 1.0 | 0.92 | 0% |
| | 2 | 49 | 16 | 16 | 26 | 26 | 69.0 | 24.26 | 33.49 | 29.16 | 35.07 | 51.52 | 50.06 | 52.04 | 45.63 | 24.57% |
| | 3 | 343 | 31 | 31 | 141 | 141 | 82.0 | 75.47 | 66.95 | 76.28 | 66.02 | 80.55 | 78.26 | 80.4 | 78.9 | 2.39% |
| Binomial Queue | 1 | 7 | 4 | 5 | 5 | 5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0% |
| | 2 | 49 | 11 | 15 | 22 | 22 | 41.0 | 41.0 | 23.03 | 41.0 | 26.96 | 41.0 | 30.88 | 41.0 | 31.23 | 0% |
| | 3 | 343 | 25 | 36 | 94 | 94 | 58.0 | 55.47 | 46.9 | 55.3 | 49.08 | 58.0 | 55.33 | 58.0 | 55.19 | 0% |
| Array | 1 | 7 | 6 | 6 | 6 | 6 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0% |
| | 2 | 49 | 19 | 19 | 31 | 31 | 10.0 | 9.25 | 8.36 | 9.33 | 8.18 | 10.0 | 9.55 | 10.0 | 9.65 | 0% |
| | 3 | 343 | 43 | 43 | 156 | 156 | 18.0 | 17.09 | 14.67 | 17.12 | 14.64 | 18.0 | 17.81 | 18.0 | 17.75 | 0% |
| AVL | 1 | 7 | 5 | 5 | 5 | 5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0% |
| | 2 | 49 | 15 | 15 | 22 | 22 | 58.0 | 58.0 | 21.2 | 58.0 | 18.82 | 58.0 | 27.88 | 58.0 | 29.96 | 0% |
| | 3 | 343 | 38 | 39 | 97 | 97 | 60.0 | 58.88 | 52.02 | 58.84 | 51.72 | 60.0 | 58.54 | 60.0 | 58.54 | 0% |
| Buffer | 1 | 8 | 1 | 1 | 1 | 1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0% |
| | 2 | 64 | 7 | 7 | 8 | 8 | 108.0 | 98.52 | 74.08 | 98.06 | 75.35 | 98.59 | 80.62 | 98.62 | 82.27 | 8.68% |
| | 3 | 512 | 17 | 17 | 38 | 38 | 132.0 | 129.56 | 125.1 | 129.73 | 124.5 | 131.93 | 130.51 | 131.93 | 130.1 | 1.51% |
| Node | 1 | 14 | 11 | 11 | 11 | 11 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0% |
| | 2 | 196 | 48 | 60 | 114 | 114 | 6.0 | 1.21 | 3.01 | 1.27 | 2.9 | 3.86 | 4.86 | 3.86 | 4.95 | 35.66% |
| | 3 | 2744 | 122 | 221 | 1203 | 1203 | 8.0 | 5.58 | 5.94 | 5.8 | 6.44 | 6.86 | 7.59 | 6.83 | 7.53 | 14.62% |
| Binary Search Tree | 1 | 8 | 6 | 6 | 6 | 6 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0% |
| | 2 | 64 | 24 | 24 | 35 | 35 | 17.0 | 15.8 | 11.76 | 15.8 | 11.28 | 15.84 | 14.47 | 15.8 | 14.16 | 7.05% |
| | 3 | 512 | 82 | 94 | 214 | 214 | 80.0 | 63.26 | 44.92 | 66.52 | 48.42 | 79.52 | 64.78 | 79.5 | 64.43 | 0.6% |
| Bounded Stack | 1 | 8 | 4 | 4 | 4 | 4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0% |
| | 2 | 64 | 10 | 10 | 14 | 14 | 69.0 | 35.31 | 27.39 | 34.51 | 28.43 | 42.0 | 34.6 | 40.97 | 36.55 | 40.62% |
| | 3 | 512 | 20 | 20 | 48 | 48 | 105.0 | 64.78 | 64.02 | 65.1 | 62.1 | 79.83 | 78.89 | 79.59 | 77.14 | 24.2% |
| Binomial Heap | 1 | 8 | 5 | 5 | 5 | 5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0% |
| | 2 | 64 | 13 | 13 | 21 | 21 | 29.0 | 22.84 | 19.84 | 22.98 | 19.09 | 24.53 | 24.39 | 24.46 | 24.56 | 15.65% |
| | 3 | 512 | 22 | 24 | 85 | 85 | 51.0 | 32.28 | 32.87 | 31.33 | 34.95 | 40.82 | 43.59 | 40.24 | 44.73 | 21.09% |
| Red Black Tree | 1 | 8 | 6 | 6 | 6 | 6 | 5.0 | 3.0 | 3.11 | 3.0 | 3.16 | 3.0 | 3.4 | 3.0 | 2.95 | 40% |
| | 2 | 64 | 21 | 22 | 33 | 33 | 17.0 | 17.0 | 17.0 | 17.0 | 16.88 | 17.0 | 17.0 | 17.0 | 17.0 | 0% |
| | 3 | 512 | 61 | 73 | 184 | 184 | 20.0 | 19.07 | 18.23 | 19.06 | 18.48 | 19.11 | 19.24 | 19.12 | 19.13 | 4.4% |
| Queue | 1 | 7 | 4 | 4 | 4 | 4 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0% |
| | 2 | 49 | 8 | 8 | 13 | 13 | 32.0 | 19.83 | 12.34 | 20.2 | 11.72 | 21.25 | 17.47 | 21.72 | 16.67 | 32.12% |
| | 3 | 343 | 10 | 12 | 40 | 40 | 53.0 | 32.58 | 27.12 | 34.11 | 29.73 | 41.13 | 40.33 | 41.76 | 40.64 | 21.2% |

Figure 11.2: Size of reduced test suites and average numbers of killed mutants

kill a mutant we apply the traditional method which consists in comparing the mutant outputs to the original program outputs.

*To compute the reduced test suites*, we first execute the original test suites and collect the tags. From these data, we build the equivalence classes w.r.t. to the four equivalence relations. Then, we select randomly one test case by equivalence class to build a tag-based-reduced test suite. The sizes of the original and reduced test suites are given in Fig. 11.2. Because the reduced test suite can be affected by the random choice of its elements in the equivalence classes, we performed the reduction 100 times and computed the average number of faults detected by each of these 100 reduced suites. Please note that the number of equivalence classes, and hence the size of the reduced test suites, is not affected by the random selection, because classes are computed before the random selection.

### 11.2.2 Results of the experiments

Fig. 11.2 gives the results of our experiments. Each line corresponds to one of the 11 systems under test (SUT), and one of the three associated test suites. Each original test suite is denoted as $TS0$.

The first five columns give the size of the original test suite and of the reduced ones, according to the four equivalence criteria. The following columns evaluate the effectiveness of the reduced test suites and compare them to randomly generated reductions of the same size. Column TB-TS, resp. R-TS, gives the average number of mutants killed by the tag-based, resp. randomly, reduced test suite.

#### Impact of the equivalence criterion on the size of the test suite

We observe that the reduction rates reach up to 90% even for the strongest relation ($R_3$). As expected, the test suite size increases for stronger relations. In some cases, the size of the test suite reduced using $R_0$ is equal to the one reduced using $R_1$. This is due to the small size of the test suites or/and due to the tags shared between operation calls. When two operations calls in two different tests produce the same tag names, the sets of tags generated by calls (used to apply $R_1$) will be equal to the set of tags generated by the test case (used to apply $R_0$).

The sizes of the reduced test suites using $R_2$ and $R_3$ are equal in these case studies. Indeed, $R_2$ and $R_3$ produce different equivalence classes when it is possible to observe two different sequences of the same set of tags. This corresponds to a different ordering of the same tags, or different iterations of some tags, like in [tag1, tag2, tag3] and [tag2, tag1, tag3] or [tag1, tag1, tag2,

tag3]. This does not appear in our experimentations because of two main points:

- Some programs do not contain constructs that create the difference in reduction rate between $R_2$ and $R_3$, as the loop construction or the recursion. When tags are inserted inside a loop, it is possible to cover the same set of tags several times and in various ordering. *Buffer* and *Array* are examples of such program.

- The test suites used for the experimentation contain test cases having small size (from 1 to 3 operation calls). The input values used for the experimentation and the length of the test cases are two essential parameters to get different tags coverage.

In the section 11.2.3, we present another experimentation aiming to show different results in reduction rate and fault detection capability between $R_2$ and $R_3$.

#### Impact of the equivalence criterion on the effectiveness of the test suite

Looking at the scores of the tag-based reduced test suites, we notice that, in most cases, stronger equivalence relations produce test suites which kill more mutants. Eight cases (out of 132) do not follow this rule. For example, considering the third test suite of AVL, $R_0$ leads to kill 58.88 mutants, while $R_1$ corresponds to 58.84. We believe that this is due to the random selection of test cases in an equivalence class.

Comparing the mutations scores of the reduced suites to the mutation score of the original one, we notice that reduced test suites may loose fault detection power. The last column of Fig. 11.2 compares the number of mutants killed by TS0 with the number of mutants killed by the tag-based reduced test suite, using $R_3$. In most cases, the difference is small, but in some cases it can rise up to 40%. Reduction can thus lead to loss of fault detection power.

Finally, we compare tag-based vs randomly reduced suites. The table gives 132 cases, corresponding to the 33 original suites and 4 different criteria. From this table, it can be noticed that our reduction strategy is better than the random one 71 times out of 132, and gives the same result in 39 cases. A statistical analysis, applying the Mann-Whitney U test with 95% confidence, confirmed that our reduction approach is better than the random one.

In summary, these preliminary experiments show that:

- stronger equivalence relations lead to larger reduced suites;

- the reduced suite corresponding to a stronger equivalence relation detects more faults than the one reduced with a weaker criterion;
- tag-based reduced suites detect more faults than randomly reduced suites of the same size.

#### Threats to validity

Since these experiments remain preliminary, they are subject to several threats to validity. As with all empirical studies involving software artifacts, threats to *external validity* questions whether the subjects used in our experiments are representative of real programs and real test suites. Our experiment concerns 33 test suites generated for classes collected on Internet. We focus on classes that include a state. To decrease the risk to select inappropriate or too specific programs under test, we consider codes that were frequently used in the literature, such as containers [Arcuri 2010], but also other types of classes [Li 2009]. Some of these classes under test (e.g. RedBlackTree) include other classes. Original test suites aimed at testing all methods in the classes contrary to the study in [Arcuri 2010], where only add and remove methods were considered.

Threats to *internal validity* correspond to bias in our experimental setting. Here, a threat to internal validity is due to the faults we considered. About the choice of faults, we apply mutation analysis, which is considered to be an unbiased and varied manner to obtain faulty programs [Arcuri 2010].

One threat to *construct validity* is the way test suites are built. To build test suites, we use a combinatorial tool. Using different tools could conduct to different test suites.

### 11.2.3 Other experimentation

We perform another experimentation aiming at observing different reduction rates between the equivalence relations $R_2$ and $R_3$. In the previous experimentation, it can be seen that the reduction rate is always the same for $R_2$ and $R_3$. This is because equivalent test cases (according to $R_2$ and $R_3$), cover the same sequence of tags sequence (equivalent according to $R_3$). We assume that this result is due the following reasons:

- Some program methods called in the experimented test suites do not contain recursion or iteration process, that make it possible to generate different tags sequence.
- The number of methods called in a test case (from 1 to 3) is not sufficient to get different sequences of tags sequence. This can explained for

example in the programs that implement the tree structure. We have to call the `insert` operation many times to execute the recursion process and get different tags. This corresponds to different kinds of insertion (such as inserting to left node, or to right node of the tree).

- The values used in the operation call may have an influence in activating different tags and then getting a different sequence of tags. For example, for a binary search tree, to insert a value we compare if its is greater than or less than the root node to insert it either to left subtree or to right subtree.

To confirm these reasons, in this experimentation, we used only the programs that contain the methods with iteration or recursion process. For every chosen program, we generate two test suites. One test suite generated combinatorially and the second is generated randomly.

The combinatorial test suites are generated using Tobias. To do that, we define test patterns that aim to test especially the methods with iteration or recursion constructs. Sets of values are defined for the operation parameters. We define also repetition of operations between m and n (`{m,n}`) with n > m. Performing this we get test cases with different sizes and diverse values. We give for example the test pattern created to test the `RedBlackTree` program:

```
group RedBlackTreeOrigPattern [us=true] {
RedBlackTree rbt = new RedBlackTree();
  (rbt.insert([10,-1,0,789655])| rbt.makeEmpty()| rbt.printTree()){4,5};
}
```

It represents a choice between three method calls repeated from 4 to 5 times: the `insert` method to insert diverse values to the tree, the `makeEmpty` method to delete all the elements in the tree and the `printTree` method to traverse the tree nodes and print its elements. The methods `insert` and `printTree` contain the recursion construct.

The random test suites are generated using the Randoop tool [Pacheco 2007]. It is a tool that generates randomly a sequence of operation calls for Java in JUnit format. To generate the randoop test suite, we have to define some parameters as the maximum length of a generated test case and the maximum number of generated tests. We choose as 50 the maximum length of a generated test case and as 5000 as the maximum number of tests generated in a single file. Randoop uses the technique of *feedback-directed random testing* to generate tests. The principle of this technique is to execute test inputs before they are recorded to avoid redundant and illegal inputs. Randoop generates randomly and in smart way a sequence of operation calls and constructor invocations of the class under test. The sequences are created incrementally, and operation calls arguments are selected

| Program | | \|TS0\| | \|TSR$_0$\| | \|TSR$_1$\| | \|TSR$_2$\| | \|TSR$_3$\| | \|TSR$_2$'\| | \|TSR$_3$'\| |
|---|---|---|---|---|---|---|---|---|
| AVL Tree | TR | 4842 | 398 | 863 | 3754 | 3758 | 2929 | 2933 |
| | TC | 5120 | 10 | 242 | 1966 | 2584 | 1659 | 2163 |
| Binomial Queue | TR | 4951 | 381 | 2886 | 4159 | 4192 | 3220 | 3250 |
| | TC | 4536 | 13 | 132 | 179 | 199 | 139 | 159 |
| Red Black Tree | TR | 5000 | 97 | 822 | 4781 | 4791 | 4415 | 4424 |
| | TC | 9072 | 106 | 461 | 1190 | 1560 | 979 | 1313 |
| Binary Search Tree | TR | 5000 | 42 | 1367 | 4851 | 4852 | 4588 | 4589 |
| | TC | 2744 | 32 | 132 | 410 | 412 | 349 | 351 |

Figure 11.3: Reduction of combinatorial and random test suites for programs with iteration or recursion process

from sequences previously constructed. The generated test suites have random length for the test cases and random values for the operation parameters. Randoop proposes also that the user adds its own values for parameter types. We used then the MAX_INT value (maximum integer) and the MIN_INT value (minimum integer) to add the upper and the lower bounds of Java int parameter.

We use the standard algorithm of reduction using the four equivalence relations ($R_0$, $R_1$, $R_2$ and $R_3$). We also apply the proposed extension that consists in deleting prefixes, to get results for equivalence relations $R_2$' and $R_3$'. The details of the extension is presented in Sect. 10.5.

The result of test suites reduction using the four equivalence relations ($R_0$, $R_1$, $R_2$ and $R_3$) is presented in Fig. 11.3. We denote the combinatorial test suite by TC and the random one by TR. The result of reduction by applying $R_2$' and $R_3$' is also presented in Fig. 11.3. We remind that $R_2$' and $R_3$' consists in reducing the test suite using $R_2$ and $R_3$ after that deleting the prefixes.

The first point we notice in the result is that the objective of this experimentation is reached by having different reduction rate for $R_2$ and $R_3$. Therefore, we can confirm the reasons identified above. It can be seen clearly that the reduction rate decreases with the strength of the equivalence relation. For example for AVL program, the reduction rate for $R_0$ is 91%, for $R_1$ is 82%, for $R_2$ and $R_3$ is 22%.

We can observe in the results, that the reduction rate of combinatorial test suites is always greater than the reduction rate of random generated ones. For example, for Binomial Queue program, the reduction rate of combinatorial test suite reduced according to R3 is 96%. For random test suite, the reduction rate is 28%. This is because the test pattern contains input values we consider diverse, are actually generating tests that cover the same system behaviors. However Random test suites contains random values, random test case size, and random operation calls making more diverse the test suites.

Looking at the reduction according to $R_2$' and $R_3$', we can notice that the extension introduced in the algorithm allows to decrease further the number of tests compared to the result of reduction according to $R_2$ and $R_3$. For example, for BinomialQueue program, the result of reduction according to $R_2$ is reduced from 4159 to 3220 ($R_2$') by deleting the prefixes, that represents an improvement reduction rate of 22%.

We can conclude by observing these results that our reduction approach is performing efficiently with randomly and combinatorially generated tests.

## 11.3 Case study: Video-On-Demand Player

In this second case study, we wanted to experiment our reduction technique using another kind of annotations: traceability annotations. Actually, there does not exist many freely available case studies which feature traceability links between requirements and code. In the best cases, links correspond to the whole class or the whole method. They don't distinguish between normal and exceptional behaviors in a method. Therefore this study is limited to one program: the Video-On-Demand Player.

### 11.3.1 Subjects

It consists of a Video-On-Demand Player, aimed to play short movie clips stored in a server. In [Lopez-Herrejon 2011], the original program was structured as a product line whose features correspond to each functional requirement. A feature is defined as a set of fragments in the code (attributes, methods or instructions). The system was created using the Feature House tool[2]. This tool implements an approach to create a software system by composing software features. The system contains 13 requirements (See Fig. 11.4), including 3 non-functional requirements (5,6 and 7). They correspond to 10 features in the code (one for each functional requirement). The whole application counts about 3000 Loc.

We constructed 4 test suites, using the combinatorial Tobias tool [Ledru 2004]. Each test suite invokes operations calls covering several requirements. Some methods are called several times, using combinations of values for each operation parameter.

Applying these tests, we discovered some subtle bugs in the application, due to an incorrect use of the AWT graphics library of Java. Some incorrect handling of multi-threading leads the program to not release some resources at the end of each test. As a result, we were unable to play large test suites

[2]http://www.fosd.de/fh

| Requirement description | |
|---|---|
| 1- Display a list of movies and select one | 8- Provide VCR-like user interface |
| 2- Play movie immediately after selection | 9- Stop a movie |
| 3- Display textual movie information | 10- Start a movie |
| 4- Pause a movie | 11- Change server |
| 5- 3 seconds max to load movie list | 12- Exit the player |
| 6- 3 seconds max to load movie textual | 13- Start the movie player |
| 7- 1 second max to start playing a movie | |

Figure 11.4: VOD Player system requirements

in JUnit, and these had to be played in pieces. This prevented us from using mutation analysis in the evaluation of test suite effectiveness. Instead, we measured code coverage (method and line coverage) for the system main classes using the Emma tool[3].

### 11.3.2 Annotation process

A feature is described in our case study as a set of code fragments that can be located in different method definitions. A tag is inserted after each fragment. For example, the feature playing a movie contains four tags corresponding to four code fragments:

- fragment of code that initializes the elements to play a movie (the attributes),
- method to play a movie,
- fragment of method to change the label of the button (to stop),
- method to destroy the thread of playing when the application is quited.

### 11.3.3 Results of the experiments

We used Tobias tool to generate 4 test suites from user-defined test patterns. These patterns allow to combine execution of successive functionalities (features) possibly with different values for the operation parameters. For example, the test suite TS2 is generated from the test pattern sq2 defined in Fig. 11.6. It allows to execute successively the features SelectMovie, PlayImmTest, StartMovie and StopMovie (the features presented in Fig. 11.4). To start a

[3]http://emma.sourceforge.net/

| | | Original | | $R_0$ | | $R_1$ | | $R_2$ | | $R_3$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Test suite | Class | Method | Line | Method | Line | Method | Line | Method | Line | Method | Line |
| $TS_1$ | VODClient | 66% | 78% | 62% | 74% | 66% | 78% | 66% | 78% | 66% | 78% |
| | ServerReq | 100% | 71% | 100% | 71% | 100% | 71% | 100% | 71% | 100% | 71% |
| | ServerSelect | 60% | 75% | 60% | 75% | 60% | 75% | 60% | 75% | 60% | 75% |
| | ListFrame | 93% | 92% | 93% | 92% | 93% | 92% | 93% | 92% | 93% | 92% |
| | Detail | 100% | 97% | 100% | 97% | 100% | 97% | 100% | 97% | 100% | 97% |
| Number of tests | | 35 | | 16 | | 16 | | 17 | | 17 | |
| $TS_2$ | VODClient | 72% | 81% | 72% | 81% | 72% | 81% | 72% | 81% | 72% | 81% |
| | ServerReq | 100% | 69% | 100% | 69% | 100% | 69% | 100% | 69% | 100% | 69% |
| | ServerSelect | 40% | 68% | 40% | 68% | 40% | 68% | 40% | 68% | 40% | 68% |
| | ListFrame | 73% | 78% | 73% | 78% | 73% | 78% | 73% | 78% | 73% | 78% |
| | Detail | 50% | 76% | 50% | 76% | 50% | 76% | 50% | 76% | 50% | 76% |
| Number of tests | | 144 | | 3 | | 3 | | 3 | | 3 | |
| $TS_3$ | VODClient | 72% | 83% | 72% | 83% | 72% | 83% | 72% | 83% | 72% | 83% |
| | ServerReq | 100% | 69% | 100% | 69% | 100% | 69% | 100% | 69% | 100% | 69% |
| | ServerSelect | 40% | 68% | 40% | 68% | 40% | 68% | 40% | 68% | 40% | 68% |
| | ListFrame | 80% | 81% | 80% | 81% | 80% | 81% | 80% | 81% | 80% | 81% |
| | Detail | 50% | 76% | 50% | 76% | 50% | 76% | 50% | 76% | 50% | 76% |
| Number of tests | | 99 | | 11 | | 11 | | 13 | | 13 | |
| $TS_4$ | VODClient | 72% | 83% | 72% | 82% | 72% | 82% | 72% | 83% | 72% | 83% |
| | ServerReq | 100% | 69% | 100% | 69% | 100% | 69% | 100% | 69% | 100% | 69% |
| | ServerSelect | 40% | 68% | 40% | 68% | 40% | 68% | 40% | 68% | 40% | 68% |
| | ListFrame | 73% | 78% | 73% | 78% | 73% | 78% | 73% | 78% | 73% | 78% |
| | Detail | 50% | 76% | 50% | 76% | 50% | 76% | 50% | 76% | 50% | 76% |
| Number of tests | | 432 | | 4 | | 7 | | 14 | | 14 | |

Figure 11.5: VOD Player evaluation results

movie (in StartMovie group), a movie have to be selected from a specified group of values (MovieTitle) where "THIS_FILM_DOESNT_EXIST" is an invalid value. Fig. 11.5 shows the results of this case study. The four original

```
group sq2{
@SelectMovieTest;
@PlayImmTest;
@StartMovieTest;
@StopMovieTest;
}

group StartMovieTest{
@StartMovie;
...
}

group StartMovie{
vodClient.setPlayImm(false);
filmTitle = @MovieTitle;
vodClient.selectmovie(filmTitle);
actionEvent = new ActionEvent(vodClient, 1, "Play movie");
vodClient.buttonControl2_actionPerformed(actionEvent);
}

group MovieTitle{
values = ["A Grand Day Out (1)", "The Wrong Trousers (1)",
"THIS_FILM_DOESNT_EXIST"];
}
```

Figure 11.6: Example of test pattern defined for the Video-On-Demand case study

test suites (TSi) were reduced using the tag-based approach. We only performed the reduction once for each criterion. For each of the five main classes of the application, we measured method and line coverage, considering the original test suites and their reduced versions. Fig. 11.5 also gives the size of each test suite.

As for our previous experiments, we notice that the size of the reduced test suites grows with stronger equivalence criteria. We can observe that the test suite reduction rate reaches up to 99% even for the strongest relation ($R_3$).

Regarding test effectiveness, code coverage is preserved by even the weakest relation ($R_0$) except for 3 cases: coverage of VODClient by TS1 and TS4 using $R_0$ and by TS4 using $R_1$.

## 11.4 Experimentation with annotated specifications

We have seen previously that our approach can be applied for annotated source code, using a tag Logger system, and we have described some experimentations performed on several programs. These experimentations show that our approach gives efficient results in terms of reduction rate and fault detection capability.

Our approach can also be used in a model-based context to reduce a test suite animated on a specification. The specification should be annotated with tags, that may be used for traceability purpose or for model commenting purpose. A system for collecting tags should be available during the test case animation to generate the covered tags.

We remind that using CertifyIt tool, we are able to animate a test case on a UML/OCL specification and to get a set of covered tags for each operation call of the test case. The result of animation for a test case is a sequence of tags sets. The CertifyIt tool is not able to collect a sequence of tags covered from an operation animation, it is only able to collect the set of tags (i.e. without execution order). This result is saved only if the test case is valid, because invalid test cases are discarded from the test suite result and are not considered to test the SUT. A sequence of tags sets is used as input to our tool to perform the reduction of test cases.

In the following, we present an experimentation of our approach on a specification of an electronic purse application.

### 11.4.1 Subjects

#### Annotated specification

In this section, we consider the case study: Electronic purse application, presented in Sect. 4.3.1. We adapted it to make all tests animated on the specification valid tests. This is done by adopting a defensive style. The OCL preconditions that make the test to fail are transformed as annotated branches in the OCL postcondition and the "true" value is assigned to the preconditions. The motivation is that we wanted to generate a tag when a test fails the original precondition. Moreover we would like to consider the behavior of

```
/**@AIM: CHECK_PIN */
if (self.isOpenSess_= true and self.mode_= Mode::USE and
    self.terminal_=Terminal::PDA and self.hptry_ > 0) then
  if (self.hptry_ > 0) then
  /**@AIM: NUMBER_OF_TRIES_IS_POSITIVE */
    if (pin = self.hpc_) then
      /**@AIM: HOLDER_AUTHENTICATED */
      self.isHoldAuth_ = true and
      self.hptry_ = self.MAX_TRY
    else
      /**@AIM: HOLDER_IS_NOT_AUTHENTICATED */
      self.hptry_ = self.hptry_ - 1 and
      self.isHoldAuth_ = false and
      if (self.hptry_ = 0) then
        /**@AIM: MAX_NUMBER_OF_TRIES_REACHED */
        self.mode_ = Mode::INVALID
      else
        /**@AIM: MAX_NUMBER_OF_TRIES_IS_NOT_REACHED */
        true
      endif
    endif
  else
    /**@AIM: NUMBER_OF_TRIES_IS_NEGATIVE */
    true
  endif
  else
  /**@AIM: TAG_OP_ERROR */
  true
endif
```

Figure 11.7: Modified OCL post condition of the checkPin operation

failing original precondition as a unique behavior among operation calls. If any operation call fails its precondition it generates TAG_OP_ERROR tag. The reason we consider a unique name of tag is because if an operation fails its precondition it will not perform any action.

We give the modified OCL postcondition of the `checkPin` operation in Fig. 11.7. (c.f. the original `checkPin` with precondition in Fig. 4.7). Every branch has been annotated with a tag. The specification contains 42 tags in total.

### Test suites

In our experimentation, we produced 10 test suites from several TSLT schemas (S1 - S10) unfolding, using the Tobias tool. The test patterns

combine several system operations (`setHpc`, `setBpc`, `authBank`, `credit`, `debit`, `checkPin`) using group of values for their parameters. Iteration construct is applied for some operations. Let us consider `S1` TSLT schema:

```
group S1 [us=true] { @Personalize; @AuthentifyHolder {0,4} }
group Personalize { ep.beginSession(ADMIN);  ep.setBpc(@BankPinVal);
                          ep.setHpc(@UserPinVal); ep.endSession(); }
group BankPinVal {values=[12,45]}
group UserPinVal {values=[56,89]}
group AuthentifyHolder {ep.beginSession(PDA); ep.checkPin(@UserPinVal);}
```

*S1* produces a suite of test cases, each starting with a card personalization (defined by the group @Personalize) followed by 0 to 4 Authentication(s) of the card holder (defined by the group @AuthentifyHolder). The group @Personalize unfolds into four sub-sequences, formed by combinations of two values for the bank PIN and the other 2 values of the card holder. The group @AuthentifyHolder is unfolded into two sub-sequences, and iteration of this group between 0 and 4 times (denoted by {0,4}) provides $2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 31$ sub-sequences. Therefore S1 is unfolded into 124 tests. The sequence S1 was then reduced using the equivalence relations $R_0$, $R_1$ and $R_2$. The equivalence relation $R_3$ is not used because CertifyIt tool is collecting the tags for an operation call as a set and not a sequence.

### 11.4.2 Results and interpretation

The reduced test suites from *S1* are containing respectively 8 and 9 tests for $R_0$ and $R_1$ and 30 tests for $R_2$. The overall results for the 10 test suites studied is given Fig. 11.8.

The results also show that the test suites are greatly reduced (up 99% of the size of the initial test suite, even with $R_2$). These good results are due to the fact that the test suites are built with a combinatorial approach. Many cases are identified as potentially different in the test are actually covering the same set of tags.

It can be noticed that the result of reduction for $R_0$ and $R_1$ are close. This is explained by the nature of the model and the associated tags. $R_0$ and $R_1$ give different equivalence classes when a tag may be associated to two different subsets of tags. For example, let us consider two test cases $TC_1$ and $TC_2$ with:

```
TC1: ep.beginSession(ADMIN);  ep.setBpc(89); ep.setHpc(56);
     ep.endSession(); ep.beginSession(PDA); ep.checkPin(56);
     ep.endSession(); ep.checkPin(56);
```

| Schéma | $\|TS_o\|$ | $\|TS_{R_0}\|$ | $\|TS_{R_1}\|$ | $\|TS_{R_2}\|$ |
|---|---|---|---|---|
| S1 | 124 | 8 | 9 | 30 |
| S2 | 496 | 15 | 21 | 74 |
| S3 | 1 984 | 26 | 39 | 204 |
| S4 | 7 936 | 25 | 45 | 604 |
| S5 | 2 048 | 10 | 13 | 132 |
| S6 | 8 192 | 10 | 13 | 428 |
| S7 | 4 096 | 120 | 130 | 552 |
| S8 | 7 239 | 22 | 25 | 155 |
| S9 | 4 096 | 5 | 8 | 31 |
| S10 | 9 216 | 45 | 60 | 122 |

Figure 11.8: Size of original test suite ($TS_o$) and reduced one ($TS_{R_i}$) with $i = \{0, 1, 2\}$ for schema Sj ($j = 1 - 10$)

$\mathcal{G}(TC_1) =$

```
[[BEGIN_SESSION, SESSION_OPENED],
[SET_BANKER_PIN_CARD, BANKER_PIN_CARD_IS_SET, MODE_IS_NOT_SET_TO_USE],
[SET_HOLDER_PIN_CARD, HOLDER_PIN_CARD_IS_SET, MODE_IS_SET_TO_USE],
[END_SESSION, SESSION_ENDED],
[BEGIN_SESSION, SESSION_OPENED],
[CHECK_PIN, HOLDER_AUTHENTICATED, NUMBER_OF_TRIES_IS_POSITIVE],
[END_SESSION, SESSION_ENDED],
[CHECK_PIN, TAG_OP_ERROR]]
```

```
TC2: ep.beginSession(ADMIN);  ep.setBpc(89); ep.setHpc(56);
     ep.endSession(); ep.beginSession(PDA); ep.checkPin(56);
     ep.beginSession(PDA);
```

$\mathcal{G}(TC_2) =$

```
[[BEGIN_SESSION, SESSION_OPENED],
[SET_BANKER_PIN_CARD, BANKER_PIN_CARD_IS_SET, MODE_IS_NOT_SET_TO_USE],
[SET_HOLDER_PIN_CARD, HOLDER_PIN_CARD_IS_SET, MODE_IS_SET_TO_USE],
[END_SESSION, SESSION_ENDED],
[BEGIN_SESSION, SESSION_OPENED],
[CHECK_PIN, HOLDER_AUTHENTICATED, NUMBER_OF_TRIES_IS_POSITIVE],
[BEGIN_SESSION, TAG_OP_ERROR]]
```

We have the set of tags generated from TC and TC2:

$g_{TC_1} = g_{TC_2} =$

```
{BEGIN_SESSION, SESSION_OPENED, SET_BANKER_PIN_CARD,
```

```
BANKER_PIN_CARD_IS_SET, MODE_IS_NOT_SET_TO_USE,
SET_HOLDER_PIN_CARD, HOLDER_PIN_CARD_IS_SET,
MODE_IS_SET_TO_USE, END_SESSION, SESSION_ENDED,
CHECK_PIN, HOLDER_AUTHENTICATED, NUMBER_OF_TRIES_IS_POSITIVE,
TAG_OP_ERROR}
```

Therefore $TC_1$ and $TC_2$ are equivalent according to $R_0$, however they don't generate the same sets of tags sets, because `TAG_OP_ERROR` appears in two subsets with a different tag. In $TC_1$, it appears with `CHECK_PIN` and in $TC_2$ it appears with BEGIN_SESSION. These two different subsets make the diversity between $TC_1$ and $TC_2$ according to $R_1$.

We identified two cases where we get a difference between a two sets of tags sets are:

- When identical tags are shared by different methods. This is the case of the example described above. The operations share the tag TAG `TAG_OP_ERROR` that correspond to a failing precondition.
- When the code (or postconditions) of the method contains several "if .. then .. else .." not nested. This may allows a tag to be generated in different operation calls with different tags. Our specification does not contain such kind of constructs.

### 11.4.3 Application of our approach on ECinema and Global Platform case studies

In this section, we present some experimentations of our approach on E-Cinema and Global Platform case studies. We experiment test suites unfolded from test patterns that are generated from test properties. The Model-based filtering approach (using filtering keys) is applied on these test patterns to resolve explosive unfolding problem. The valid test cases result are reduced using the three equivalence relations $R_0$, $R_1$ and $R_2$. We reuse the same patterns generated in Sect. 6.6 and Sect. 6.7. In Fig. 11.9 we present the result of test suite reduction. We can see that the reduction rate can reach 92%. We observe also that the reduction rate decreases with the strength of the equivalence relation.

Applying our approach on the annotated specification of Global Platform generates a reduced test suite whose size is more likely to run on the card. In the example of sc_alpha3_temp1 test pattern, there are 7 representative tests to run on the card instead of 1117.

| Case study | Test pattern | \|T0\| | $\mid T0_{R_0} \mid$ | $\mid T0_{R_1} \mid$ | $\mid T0_{R_2} \mid$ |
|---|---|---|---|---|---|
| ECinema | sc1_prop2 | 702 | 25 | 25 | 25 |
| | sc1_prop3 | 132 | 16 | 16 | 60 |
| | sc1_prop3_bis | 840 | 21 | 21 | 120 |
| Global Platform | sc_alpha3_temp1 | 1117 | 7 | 7 | 7 |

Figure 11.9: Results of test suite reduction in ECinema and Global Platform case studies

## 11.5 Conclusion

In this chapter, we presented experimentations of our test suite reduction approach on several case studies. These case studies are defined in different contexts. We used Java applications where tags are inserted for source code instrumentation. We report also on another large case study: VOD movie player, a graphical Java application where a 3000 lines program was tagged to trace 10 functional requirements. Our approach is also applied for specifications where tags are inserted to trace user requirements. The specification of EPurse, ECinema and Global Platform case studies are used.

We use several test suites generated from a combinatorial approach (Tobias) or from a random approach (Randoop). We apply on these test suites the reduction technique using the four equivalence relation (R0, R1, R2, R3). We use several criteria to evaluate the efficiency of our approach: reduction rate, code coverage and fault detection capability. The experimentation results on small Java applications shows that the stricter equivalence relations lead to a better fault detection power of the reduced test suite. We have also shown that the reduced test suites detect more faults than randomly reduced ones of the same size. The results shows good reduction rate that can reach 99% even for the strictest equivalence relation. It also shows that the reduction rate increases with the discrimination power of the equivalence relation. This result on reduction rate is also true for annotated specification of EPurse, ECinema and Global Platform case studies. Using the VOD player case study, code coverage of the initial and reduced suites were compared. The case study suggests that significant size reduction can be achieved from this tagging while keeping a satisfactory level of code coverage. We have also experimented our approach using the proposed extension of our algorithm that consists to improve the reduction rate of the reduction algorithm by deleting prefixes. It has been shown in the result that the reduction rate is improved by keeping the same fault detection capability. Observing the results shown by the experimentations, we can conclude that our reduction approach can be

used efficiently to reduce a test suite relying on tags inserted in source code or specification. One can choose the equivalence relation the most appropriate depending on the testing needs. If we would like to generate a reduced test suite with good quality, we have to choose a strong equivalence relation. However, if it is suggested that the reduction rate is more important in a specific context, we have then to choose a weak equivalence relation.

We already mentioned the potential diversity of the tagging approaches. Further work should explore this diversity more systematically, addressing questions such as the reuse of tags at several places, the granularity of tagging, and the appropriate level for requirements traceability. Further work also includes additional experimental evaluation. It would be interesting to compare our approach to classical reduction techniques based on code coverage. Actually, the annotation scheme used in these case studies is close to the measurement of code coverage. We expect that the resulting reduced test suites will have similar characteristics as the ones produced from code coverage information. Further experiments are needed to confirm this conjecture. Other works may explore the definition of additional equivalence relations, as well as evolutions of the algorithm. A simple evolution of the algorithm would select more test cases if the equivalence class is more populated. This could produce reduced test suites which are more representative of the original test suite diversity.

Chapter 12

# Summary of test suite reduction contribution

## 12.1 Motivation

Our contribution is proposed in the context where a large number of tests is generated automatically or defined manually. These tests are used to explore maximum number of the behaviors of the SUT to detect failures. However, this large number of tests can not be executed on the SUT when resources are limited (time, system CPU or memory), for example when we test smart card applications, or in the context of regression testing, where new tests are added to the test suite to test new or changed requirements.

## 12.2 Tags and annotation process

Our test suite reduction approach relies on coverage information denoted as tags. These tags are inserted in the source code or the specification of the SUT.

Tags are textual information inserted in the form of comments or executable instructions. They can be inserted manually or automatically for source code instrumentation purpose or for traceability purpose. When they are inserted to instrument the code, they can be used for example to measure the code coverage. If they are inserted for traceability purpose, they can be used to trace back a user requirement. Whatever the purpose, we take advantage of these inserted tags to realize the test suite reduction.

The way our code or specification is annotated may affect our reduction result. In the experimentation of Java programs, we have a manual tagging process, we choose to tag the points that affect the result or change the system state. For example, tagging a conditional branch where an attribute value is modified. We choose also to insert tags after the related set of instructions to be sure that all of them are executed before covering the tag. A tag can be unique or not if we see that two regions of code are performing the same process. Another mode of tagging can be proposed in the context of requirement traceability, where a requirement in a model has to be linked

to requirement in the code. However, in the literature this process is still in progress, and tags inserted in code are limited to the classes, fields and methods. Tagging a fine grained requirement defined as a set of instructions inside a method call is not suggested.

Like other test suite reduction technique based on code coverage, the test suite to reduce have to be executed at least one time to get the covered tags. For annotated source code, we developed a tag logger system that logs a tag to a file once covered.

For annotated specifications, we use an animator engine that animates a test on the specification and gets the activated tags. In our work, we use CertifyIt tool as a test animator on UML/OCL specifications.

## 12.3 Equivalence relations and reduction algorithm

Using the covered tags, the idea of our contribution is to reduce the test suite based on equivalence. When two tests are equivalent according to some equivalent relation, we consider that only one is sufficient for testing and remove one of the two from the test suite. Using this idea, we get some diversity among tests in the reduced test suite, and we consider this reduced test suite as representative of the original one.

When we execute a test suite to get the covered tags, it results for each test case a sequence (1) of sequences (2) of tags. The sequences (2) correspond to the sequences of tags generated from operation calls executed in the sequence (1) of operation calls. From the result generated, we have proposed to construct a family of equivalence relations, taking into account (or not) the order and repetition of tags covered. The order and the repetition of tags can be considered inside an operation call or between two operation calls in the sequence. In the case where we consider them *inside* an operation call, the order and the number of times a set of instructions is executed (denoted by the tag) are important and may have an influence. In the case we consider them *between* two operation calls, the order and the number of times an operation call is executed are important in the sequence of operation calls.

Considering on these two criteria (order and repetition of tags) we propose four equivalence relations (R0, R1, R2 and R3), from weakest to strongest one. If the order and repetition of tags are considered, we compare the sequence of tags, otherwise we compare the set of tags between two test cases. The weakest relation means that the order and the repetition of tags are not considered. The strongest one requires that two test cases are equivalent if they cover exactly the same sequence of tags sequences.

The algorithm of reduction is applied using the coverage tags of each test case and a chosen equivalence relation. Sample examples show that the reduction rate decreases with the strength of the equivalence relation.

An extension of algorithm has been proposed to improve the reduction rate. This extension consider a test case as a prefix if its sequence of tags sequence is included in another sequence of tags sequence of another test case. The extension consists then in finding and deleting the prefixes in the test suite.

## 12.4 Case studies

Three different case studies have been performed to evaluate our approach. We consider annotated code and annotated specification. Two case studies were provided externally to our team: from third party (our project partners or research colleagues). They represent code and specification annotated to trace requirements. The results show good performance of our approach in terms of reduction rate that can reach more than 90%.

We also evaluate the fault detection capability of our approach by using mutation testing. The results show also good performance by killing with the reduced test suite almost the same mutants killed by the original test suite even using the weakest equivalence relation. The number of mutants killed increases with the strength of the equivalence relation, however the reduction rate decreases with the strength of the equivalence relation.

## 12.5 Conclusion and perspectives

We conclude that our contribution can be used efficiently to produce a reduced test suite representative of the original one in terms of its capability to detect fault. The reduction rate is very high for weakest equivalence relation and acceptable for strongest one. The choice of an equivalence relation depends on many factors. For example if the elements in the code/specification are strongly dependent in terms of result/output production, it is better to choose a strong equivalence relation that takes into account the order and the repetitions of tags.

The dependency between operations is also a factor for equivalence relation choice. By taking these factors into account we try to get a high quality test suite (representative of the original one). In some cases, the reduction rate is more important than the quality of the test suite as in the context to test a system in very limited resources. A common case where we don't have much CPU or memory resources. Another example when the time dedicated

for testing activity is limited. In this cases, a weaker equivalence relation is chosen.

As perspectives, we would like to add new equivalence relations based on the order and repetition of tags. For example, we intend to implement the equivalence relation that consider two test cases as equivalent if they cover the same set of tags sequence. We would like also to propose other kinds of relations, always using covered tags, but basing on subsumption of tags. A subsumption relation can be: if a test $t_a$ generates a sequence of tags sequence that is a prefix of a sequence of tags sequence of another test $t_b$. In this case, we can remove the test $t_a$ from the test suite. Another perspective concerns the annotation process, to experiment the effect of tagging the code of the reduction result. For example, we can evaluate the effect of inserting tags in different regions (condition, iteration, recursion, etc.) on the size of the reduced test suite.

A final perspective consists in experimenting our approach on other large programs annotated in all possible types of constructs (for example in java: if, for, while, do, switch, etc.)

Chapter 13

# Conclusion

Contents



## 13.1 Summary of the the thesis

In this thesis we presented two contributions developed to bring solutions to two main issues related to **combinatorial testing** technique.

The first issue is manifested in the *test generation time*. It consists in two sub-issues:

- A **large** number of tests unfolded from test patterns are **invalid** according to the specification. This problem occurs because combinatorial testing technique does not rely on the specification of the SUT to generate tests. Our solution consists in coupling the combinatorial technique to an *animation* technique. The tests generated combinatorially are animated on a specification of the SUT to report whether they are valid or not. Only valid tests are kept in the test set result. We call this principle as **test filtering** according to the specification.

- Some test patterns are subject to *combinatorial explosion* where a huge number of tests will be unfolded. This problem is triggered because the test pattern contains many input values. The solution proposed to fight the combinatorial explosion is to **incrementally** unfold the test pattern. It consists in processing the test pattern in iterations. In each **iteration** one (sequence of) instruction(s) is unfolded, and the tests generated are animated on the specification to filter invalid tests. The valid prefixes are combined with the (sequence of) instruction(s) of the next iteration to be unfolded. The advantage of this technique is to discard invalid tests at early stages. We can further reduce the combinations from an iteration to another by selecting a **proportion** of

valid prefixes rather than selecting all of them. Other constructs are also proposed in test patterns to filter tests in the incremental process. The **behavior filtering** construct consists in discarding tests that contain an operation call that does not cover the behavior specified by the test engineer. The **behavior** is expressed as a set of **tags** often inserted in the method branches. The **state predicate filtering** construct allows to filter tests that do not satisfy a given property at a specific point in the operation sequence. These two filtering constructs are used as **directives** to remove the tests that do not fulfill the requirements and to target the desired tests.

The first contribution represented by the solutions proposed above is implemented using **Tobias** tool as a combinatorial testing tool and **CertifyIt** tool as an animation tool. The Tobias generated tests are animated on a UML/OCL specification to filter invalid tests. However, the principles of the solutions are independent of the technologies, and they can be used with other combinatorial and animation tools.

This first contribution is illustrated in 3 main case studies. One case study was provided by our research team: the electronic purse application (**EPurse**). The test patterns in this case study are defined by us aiming to test a sequence of operation calls using some user requirements. Our incremental process and filtering constructs allow to find test cases hidden in a huge search space (e.g. $19^{18}$ tests). The two other case studies were provided by our project partners: on-line vending application of cinema ticket (**Ecinema**) and **Global Platform**, a last generation of smart cards operating system. The test patterns in these case studies are generated automatically from test properties using the tool chain of the ANR TASCCC project. These case studies show limitations of our incremental approach related to **explosive iterations**. Solutions are proposed to deal with this problem by introducing some rules to redefine test patterns and to make the incremental unfolding work and produce valid tests. Moreover, Tobias random selectors are also used to reduce the number of elements unfolded from explosive groups. We have seen that very explosive test patterns have been addressed to get valid results. For example, in Ecinema case study we get valid tests from a test pattern with $1.89 * 10^9$ tests. We also get valid tests from a test pattern that contains an explosive iteration with $3.4 * 10^{11}$ elements. In Global Platform case study, using our approach we were able to find valid tests in a search space $> 10^{100}$ tests.

The second issue is manifested in the *test execution time*. In the context of regression testing, as many test cases are added to the test suite to evaluate new or modified requirements, the size of the test suite grows rapidly

and the cost of executing it becomes more expensive. This problem can be resolved by applying a test reduction technique, it takes the large test suite and provides a reduced test suite representative of the original one. In our work, we proposed a new test reduction technique, that uses an equivalence relation based on execution **traces** of tests to reduce tests based on similarity. The traces generated consist of tags inserted in the implementation/specification and covered during execution/animation. The tags are inserted for **code instrumentation** purpose, for example to compute the code coverage. They can also be inserted for **traceability** purpose to trace back user requirements. The trace generated records the **order** and **repetition** of tags inside an operation call and inside the test case. They are related to the order of execution and the number of times the tag is covered by execution. Using the criteria of order and repetition of tags, a family of 4 **equivalence relations** is proposed to decide on equivalence of two test cases. The weakest one does not take into account the order and number of repetitions of tags and consists in comparing the set of tags generated from two tests. The strongest one requires that two equivalent tests have to record exactly the same sequence of tags.

This second contribution was experimented using many examples of test suites. It was first experimented on small programs where tags are inserted manually especially in the method branches. The original test suites were generated combinatorially and randomly, and reduced using the 4 equivalence relations. The reduced test suites were compared to the original ones and to randomly generated test suites of the same size. The comparison was performed in terms of fault detection capability (using mutation testing) and reduction rate. In these case studies, the reduction rate decreases with the strength of the equivalence relation and can reach 99%. The results also show that stricter equivalence relations lead to a better fault detection power of the reduced test suite. Moreover, reduced test suites detect more faults than randomly reduced ones of the same size. In many cases, the reduced test suites have the same fault detection capability as the original one. We also experimented our approach using some combinatorial test suites on the case study of the Video On Demand Player, where tags are inserted to trace user requirements. The results show that significant size reduction can be achieved from this tagging while keeping a satisfactory level of code coverage.

This contribution was also experimented in the model-based context where tags are inserted in UML/OCL model. Tests are animated using CertifyIt tool, saving the tags activated during animation. The results show significant reduction rate for the EPurse, ECinema and Global Platform case studies.

## 13.2 Perspectives

The perspectives of our research work concerning the two contributions include:

1. Proposing other mechanisms to filter tests in the test pattern unfolding.
2. Improving the model-based filtering algorithm by automating tasks that are performed manually.
3. Proposing other equivalence relations and subsumption relation to reduce tests. The subsumption consists in comparing the trace of tests to observe inclusion of tests in other tests.
4. Comparing our test reduction technique to the structural reduction technique by automatically inserting tags inside source code according to some structural coverage criteria.
5. Performing experimentations of the test reduction techniques on large case studies where tags are inserted by external party.

Another perspective is to further explore the constructs proposed by jSynoPSys. In many cases, it is difficult to restrict the set of operation calls, or the set of values included in a group. To solve this problem, jSynoPSys introduces a form of "wild card": $\$OP$ represents the group of all possible operation calls and $\$V$ the set of all possible values. Also $\$OP*$ represents an iteration with undefined bounds. With these constructs, it is possible to characterize the prefix of a test pattern with minimal effort by stating the state property that must be reached at the end of the prefix:
$\$OP* \rightsquigarrow$`{property to reach}`
In jSynoPSys, constraint programming techniques are used to instantiate $\$OP*$, the sequence of operations leading to the desired state. If we want to include the $\$OP$, $\$V$ constructs in Tobias, we can consider them as sets of operations or values. Moreover, to cope with the unfolding principles of Tobias, these sets must be finite.

We propose to populate these sets by observation of a repository of available test cases. These test cases may be for example the ones produced by CertifyIt. A perspective is to use pattern matching and data mining techniques to discover interesting values or instantiated operation calls or sequences.

Appendix A

# JML specification of container manager system

```
public class ContainerManager {
  private /* @ spec_public */ int cont1=0;
  private /* @ spec_public */ int cont2=0;
  private /* @ spec_public */ int cont3=0;
  private /* @ spec_public */ int cont4=0;
  /*
   * @ requires lo1>= 0 && lo2 >=0 && lo3>=0 && lo4>= 0 &&
   *(cont1+lo1)<1000 && (cont2+lo2)<3000 &&
   *(cont3+lo3)<5000 && (cont4+lo4)<7000;
   * @
   */
  public void load(int lo1, int lo2, int lo3, int lo4) {
    cont1 = cont1 + lo1; cont2 = cont2 + lo2;
    cont3 = cont3 + lo3; cont4 = cont4 + lo4;
  }
  /*
   * @ requires (lo>=0 && num>=1 && num<=4) &&
   * ((num==1 && lo<cont1)||(num==2 && lo<cont2)
   * ||(num==3 && lo<cont3)||(num==4 && lo<cont4));
   * @
   */
  public void unload(int num, int lo) {
    if (num == 1) {
      cont1 = cont1 - lo;
    } else if (num == 2) {
      cont2 = cont2 - lo;
    } else if (cont == 3) {
      cont3 = cont3 - lo;
    } else if (num == 4) {
      cont4 = cont4 - lo;
    }
  }}
```

Appendix B

# Complete description of a generated test pattern in ECinema case study

```
header{scenarioId=#sc_prop1_tagDeletion_0_0#}
group sc_prop1_robustness [us=true] {
  @sequenceGroup0{1, 1};
}

group default_boolean {
  values=[true,false];
}
group simpleOperationCall3 {
  sut.login(@default_enum_USER_NAMES, @default_enum_PASSWORDS)
            /w {set(@AIM:LOG_Success)};
}
group setMinusGroup1{
  SET = @base_call3 setMinus @call_restriction5
}
group sequenceGroup0 {
  @simpleOperationCall0{0,2};
  @simpleOperationCall1;
  @simpleOperationCall2{0,2};
  @simpleOperationCall3;
}
group default_enum_PASSWORDS {
  values=[ INVALID_PWD, PWD1, PWD2, PWD3,
           REGISTERED_PWD, UNREGISTERED_PWD];
}
group all_instances_Movie {
  values=[ film1, film2];
}
group anyCalls {
  @all_operations_ECinema
}
group all_operations_ECinema {
```

```
 ( @all_instances_ECinema.buyTicket(@default_enum_TITLES)
 | @all_instances_ECinema.checkAvailableTickets(@default_enum_TITLES)
 | @all_instances_ECinema.checkMessage()
 | @all_instances_ECinema.closeApplication()
 | @all_instances_ECinema.deleteAllTickets()
 | @all_instances_ECinema.deleteTicket(@default_enum_TITLES)
 | @all_instances_ECinema.goToHome()
 | @all_instances_ECinema.goToRegister()
 | @all_instances_ECinema.login
     (@default_enum_USER_NAMES, @default_enum_PASSWORDS)
 | @all_instances_ECinema.logout()
 | @all_instances_ECinema.registration
     (@default_enum_USER_NAMES, @default_enum_PASSWORDS)
 | @all_instances_ECinema.showBoughtTickets()
 | @all_instances_ECinema.unregister())
}
group default_enum_MSG {
 values=[ ALL_MOVIES_SOLD_OUT, ALREADY_LOGGED_IN,
          ALREADY_REGISTERED, BYE, EMPTY_PASSWORD,
          EMPTY_USERNAME, EXISTING_USER_NAME, LOGIN_FIRST,
          NONE, NO_MORE_TICKET, REGISTER, REGISTER_FIRST,
          UNKNOWN_USER_NAME_PASSWORD, WELCOME, WRONG_PASSWORD,
          WRONG_STATE];
}
group default_enum_SystemState {
  values=[ DISPLAY, REGISTER, WELCOME];
}
group simpleOperationCall2 {
  (@setMinusGroup1)
}
group simpleOperationCall1 {
  sut.buyTicket(@default_enum_TITLES);
}
group call_restriction4 {
  ( @all_instances_ECinema.login(@default_enum_USER_NAMES,
    @default_enum_PASSWORDS))
}
group simpleOperationCall0 {
  (@setMinusGroup0)
}
group call_restriction5 {
( @all_instances_ECinema.login
    (@default_enum_USER_NAMES, @default_enum_PASSWORDS))
```

```
}
group all_instances_Ticket {
  values=[ t1, t2, t3, t4, t5, t6, t10, t9, t8, t7];
}
group base_call3 {
  ( @anyCalls)
}
group call_restriction1 {
  ( @all_instances_ECinema.buyTicket(@default_enum_TITLES)
  | @all_instances_ECinema.login
     (@default_enum_USER_NAMES, @default_enum_PASSWORDS))
}
group base_call0 {
  ( @anyCalls)
}
group default_enum_TITLES {
  values=[ TITLE1, TITLE2, TITLE3, TITLE4, TITLE5];
}
group call_restriction2 {
  ( @all_instances_ECinema.buyTicket(@default_enum_TITLES)
  | @all_instances_ECinema.login
      (@default_enum_USER_NAMES, @default_enum_PASSWORDS))
}
group all_instances_ECinema {
  values=[ sut];
}
group default_enum_USER_NAMES {
  values=[ INVALID_USER, REGISTERED_USER,
           UNREGISTERED_USER, USER1, USER2, USER3];
}
group setMinusGroup0{
  SET = @base_call0 setMinus @call_restriction2
}
group all_instances_User {
  values=[ registeredUser, unregisteredUser,
           invalidUser, erronedUser];
}
```